AN EVALUATION OF ARTIFICIAL NEURAL NETWORKS

IN THE

DESIGN OF SINGLE SAMPLING PLANS

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by

ARUN KUMAR GUPTA

to the

DEPARTMENT OF INDUSTRIAL AND MANAGEMENT ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR.
APRIL, 1994.

ACKNOWLEDGEMENT

The work in this thesis has been carried out under the supervision of Dr. T.P. Bagchi. I am extremely grateful to him for the support he has extended to me during the entire duration of the thesis work.

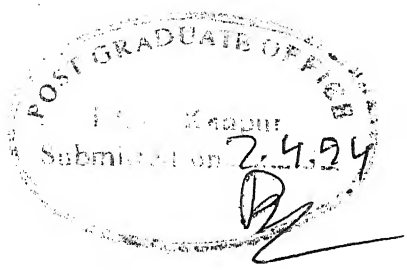
I express my deep sense of gratitude to Dr. S. Narshimhan of Chemical Engineering Department for his full support and timely advice throughout this work, without which I would say nothing could have been done, who inspite of his busy schedule found time to help me out whenever I needed his help. Thank you, Sir !.

I am grateful to Dr. P.R.K. Rao and Dr. P.K. Kalra of Electrical Engineering Department for their expert advice on the finer details of the artificial neural networks.

Finally, I would like to thank my classmates, especially Krishan Raman and Sanjeev Swami for their moral support during my hard times in pursuing this thesis.

13 April '94. A.K. Gupta

I.I.T - Kanpur.



CERTIFICATE

It is certified that the work contained in the thesis entitled *"An Evaluation of Artificial Neural Networks in the Design of Single Sampling Plans"* by *Arun Kumar Gupta*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Tapan P. Bagchi)
Professor
Industrial & Management Engineering
Indian Institute of Technology
Kanpur - 208 016

5th April, 1994

ABSTRACT

This work reports on an attempt to evaluate the applicability of artificial neural nets to guide the design of single sampling acceptance plans. Acceptance sampling, especially single sampling, constitutes a common quality assurance tool to assure that goods received in consignments or lots are of acceptance quality. A scientifically designed sampling plan attempts to balance (consumer's and producer's) risk protection with the total effort in sampling. The common methodology employed in designing single sampling plans is the Larsen nomograph — a graphical technique — with its limited popularity in the field. Artificial neural nets (or ANNs), in use now for almost a decade, have been proposed as a practical means to represent relationships among quantitative or qualitative variables, such as the sampling plan parameters. ANNs residing in a computer can learn (i.e., infer relationships) directly from data presented to them, avoiding any analytical procedures. Typical applications of ANNs have occurred in classification problems, (mathematical) function estimation, data compression, and similar tasks. In the present work "patterns" of data consisting of an input vector (AQL, $1-\alpha$, RQL, β) and the corresponding output vector (n, c) were presented to variously configured 3-layer MLPs (multi-layered perceptrons), and then the predictability (measured as the total square of errors) of the ANN was evaluated. Training utilized the backpropagation algorithm to optimize the "weights" of the MLP. Test using sets of up to 100 training patterns demonstrated that MLPs do have the potential to help in determining the single sampling plan parameters (n and c) given the QA requirements (AQL, $1-\alpha$, RQL and β) at hand. However, even 100 patterns demanding considerable computing effort did not produce sampling plans with acceptable accuracy. This study concludes that the training pattern set must be considerably expanded if one wishes to produce an ANN capable of completely replacing the Larsen nomograph.

C O N T E N T S

LIST OF FIGURES

LIST OF TABLES

A	the input layer
a_i	output from node i in the input layer
B	the first hidden layer
b_i	output from node i in first hidden layer
C	the second hidden layer
c_i	output from node i in second hidden layer
c_k^m	actual output from the kth processing element in the mth training pattern
c_n	actual output from node n
d_k^m	desired output from the kth processing element in the mth training pattern
d_n	desired output from node n
E	total squared error
e_j	jth component of the error vector
I_i	input to the ith node in the input layer
I_L	the input vector
L	number of nodes in input layer
M	number of training patterns presented to the input layer
M'	the training subset from set M
M''	the test subset from set M
m	number of nodes in the hidden layer
n	number of nodes in the output layer
T_j	internal threshold for the jth processing element
T_{Ai}, T_{1i}	internal threshold for layer A
T_{Bj}, T_{2j}	internal threshold for layer B
T_{Ck}, T_{3k}	internal threshold for layer C

v_{ij} interconnection weight between the *i*th node of layer A and the *j*th node of B

w_{ij} interconnection weight between the *i*th node of layer B and the *j*th node of C

x_k^m sum of the weighted inputs to the *k*th node on the output layer for training pattern *m*

α momentum factor

β gain factor

δ_{2j}^m gradient descent term for the *j*th processing element on the hidden layer (layer 2) for training pattern *m*

δ_{3k}^m gradient descent term for the *j*th processing element on the hidden layer (layer 3) for training pattern *m*

ϵ_n component of the output vector from the *n*th node on the output layer

η learning rate

CHAPTER 1
SENTENCING LOTS BY ACCEPTANCE SAMPLING

1.1 Acceptance Sampling

One major aspect of statistical quality control is acceptance sampling. A buyer often receives a shipment of goods in the form of a large ''lot''. Rather than doing 100% inspection of all items, the buyer frequently *samples* the shipment and then either accepts it as conforming to its standards or rejects it on the basis of the quality of the items in the sample. If the lot is judged to be below standard, it may be returned to the supplier or kept, depending on how badly the goods are needed or on what arrangements have been made with the supplier regarding their disposal. Possibly there will be a price concession on rejected lots. When third party inspectors in a supplier's plant clear all shipments, no return of lots is necessary since unacceptable lots would be detected before they are shipped. However, as a practice, most companies do not accept or return rejected lots to the supplier until they are reasonably sure through sampling inspection that the lot is of unacceptable quality. Furthermore, a company's own output may also be subjected to acceptance sampling at various stages of production. A given lot of goods may be sampled and either accepted for further processing or for shipment to customers, or rejected.

The purpose of acceptance sampling is to determine a course of action regarding the disposal of a lot (i.e., its acceptance or rejection) and not to *estimate* lot quality. Acceptance sampling prescribes a procedure that, if applied to a series of lots, assures a specified risk of accepting lots of given quality. In other words, acceptance sampling yields quality assurance.

The indirect effects of acceptance sampling on quality may often be much more important than the direct effects. When a supplier's product is being rejected at a high rate, one of two

things may happen. The supplier may take steps to improve his production methods, or the customer may be led to seek other and better sources of supply. In many cases, large companies send out their own experts to help suppliers solve quality problems. Acceptance sampling thus indirectly improves quality of production through its encouragement of good quality by a high rate of acceptance and its discouragement of poor quality by a high rate of rejection.

Furthermore, acceptance sampling when used by a manufacturer at various stages in production, generally benefits the quality of production. If a company relies heavily and only on final 100 percent inspection of the goods shipped to customers, a careless attitude toward quality may be generated among production personnel. Workers and management alike may feel that no bad material will be shipped out because all substandard material will be caught in final inspection. Emphasis in the production department may then be on quantity of output rather than quality. Final 100 percent inspection, however, may not be as effective as believed; for because of its monotonous and boring character, a good percentage of defective material may actually get by, the more the poorer the quality submitted for inspection. Even if final inspection is perfect, a poor attitude toward quality by production personnel will lead to large scrap and rework. On the other hand, under a program of acceptance sampling, the cost of screening and reworking rejected lots may be placed on the production department. Production personnel will then become quality conscious and there will be an interest in quality on the part of both inspection and production. The rule will be: "Make it right the first time" *(Duncan, 1974)*. These psychological aspects of acceptance sampling are of major importance.

Acceptance sampling is used under the following conditions:

1. When the cost of inspection is high and the loss arising from the passing of a defective unit is not great. It is possible in some cases that no inspection at all will be the

cheapest plan.

2. When 100 percent inspection is fatiguing, a carefully worked-out sampling plan will produce as good or better results. As noted above, 100 percent inspection may not mean 100 percent quality, and the percentage of defective items passed may be higher than under a scientifically designed sampling plan.

3. When inspection is destructive. In this case sampling must be employed.

1.2 Forms of Sampling

Sampling plans can be classified in two categories: *attributes plans* and *variables plans (Juran, 1988)*.

1.2.1 Attributes plans. In these plans, a sample is taken from the lot and each unit classified as conforming or non-conforming. The number nonconforming is then compared with the acceptance number stated in the plan and a decision is made to accept or reject the lot. Attributes plans can further be classified by one of the two basic criteria:

1. Plans which meet specified sampling risks provide protection on a lot-by-lot basis.Such risks are:

(a) A specified quality level for each lot (in terms of percent defective)having a selected risk (say 0.10) of being accepted by the consumer. The specified quality level is known as the lot tolerance percent defective (p_2); the selected risk is known as the consumer's risk (β).

(b) A specified quality level for each lot such that the sampling plan will accept a stated percentage (say 95 percent) of submitted lots having this quality level. This specified quality level is termed the acceptable quality level (AQL). The risk of rejecting a lot of AQL quality (p_1) is known as the producer's risk (α).

2. Plans which provide a limiting average percentage of defective items for the long run. This is known as the average outgoing quality level (AOQL).

1.2.2 Variable Plans. In these plans, a sample is taken and a measurement of a specified quality characteristic is made on each unit. These measurements are then summarized into a simple statistic (e.g.,sample mean) and the observed value compared with an allowable value defined in the plan. A decision is then made to accept or reject the lot. When applicable, variables plans provide the same degree of consumer protection as attributes plans while using considerably smaller samples.

1.3 Attributes Sampling

An overview of single, double, multiple, and sequential plans:

1.3.1 Single sampling. In single sampling by attributes, the decision to accept or reject the lot is based on the results of inspection of a single sample selected from the lot. The operation of a single sampling plan is given in figure 1.1.

1.3.2 Double sampling. In double sampling by attributes, an initial sample is taken, and a decision to accept or reject the lot is reached on the basis of this first sample if the number of nonconforming units is either quite small or quite large. A second sample is taken if the results of the first sample are not decisive. The operation of an attributes double-sampling plan is given in figure 1.2.

1.3.3 Multiple sampling. In multiple sampling by attributes, more than two samples can be taken in order to reach a decision to accept or reject the lot. The chief advantage of multiple sampling plans is a reduction in sample size for the same protection.

1.3.4 Sequential sampling. In sequential sampling, each item is treated as a sample of size one, and a determination to accept,

Single Sampling Plan Scheme

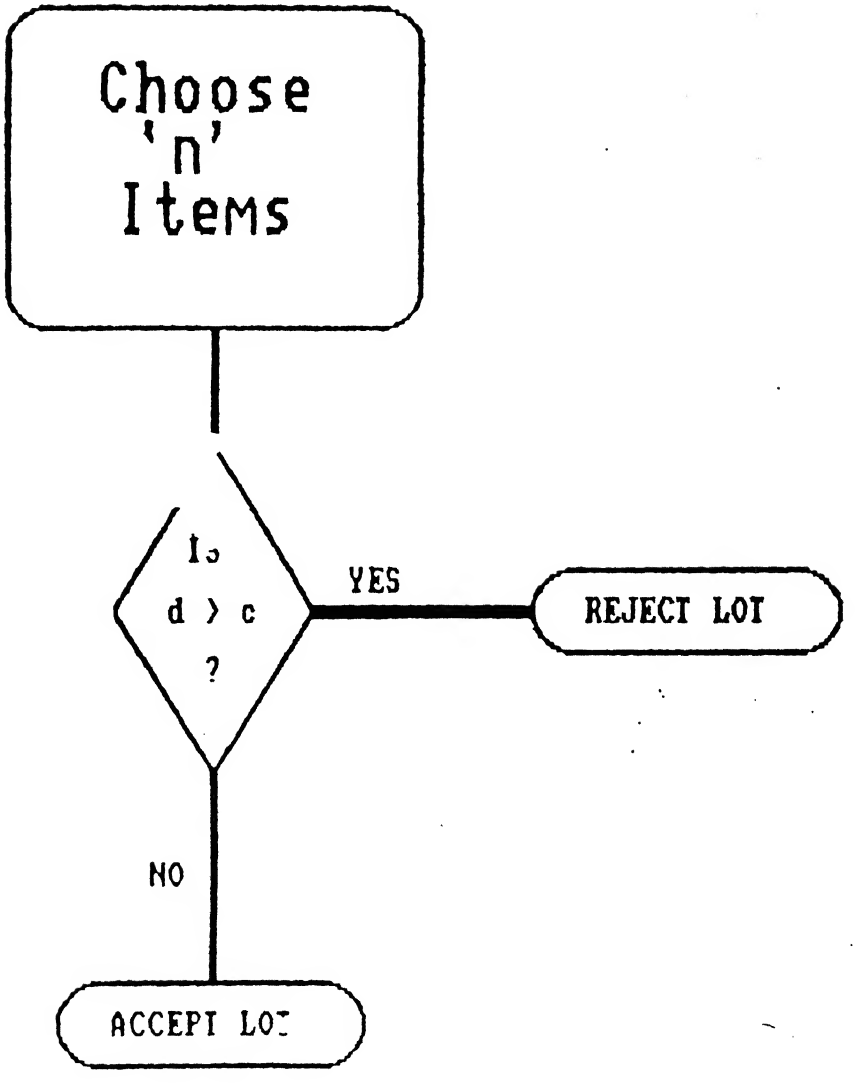


Figure 1.1

Double Sampling Scheme

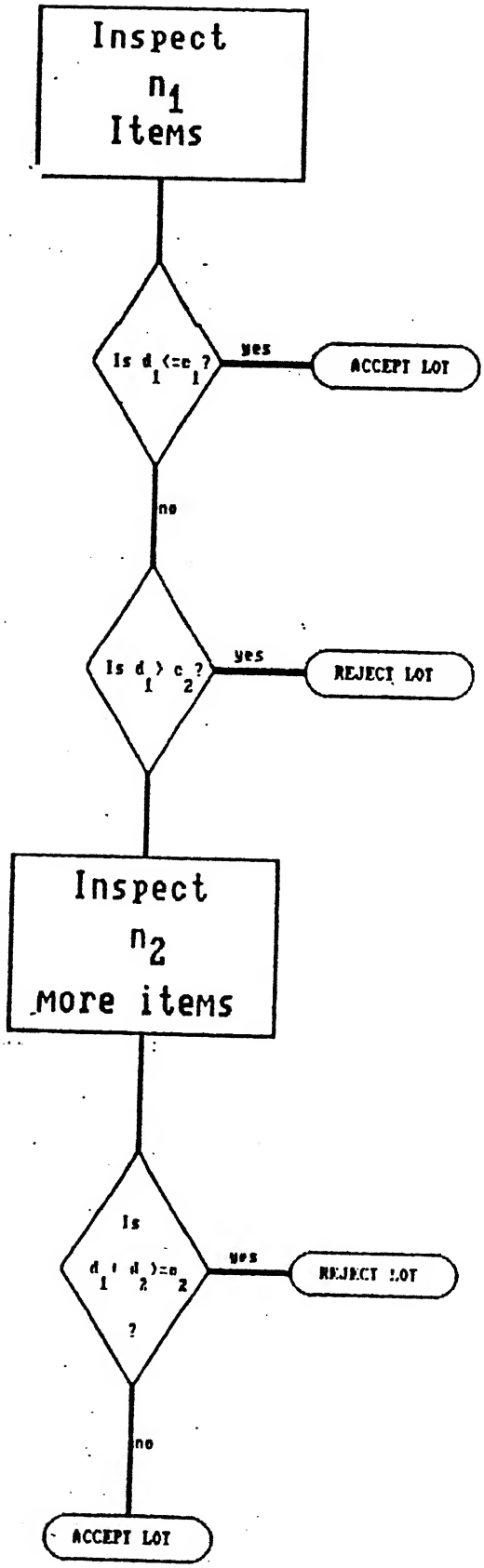


Figure 1.2

reject or continue sampling is made after inspection of each item. The major advantage of sequential sampling plans is that these plans offer the opportunity for generally achieving the minimum sample size for a given protection.

1.4 Selecting Sampling Plans

In general, all four above sampling plans can be planned to give lots of specified qualities nearly the same chance of being accepted; i.e., the operating characteristic curves can be made quite similar (matched) if desired. However, the best type of plan for one producer or product is not necessarily best for another. The suitability of a plan can be judged by considering the following factors:

1. Average number of parts inspected
2. Cost of administration
3. Information obtained as to lot quality
4. Acceptability of plan to producers.

The average number of parts that need to be inspected to arrive at a decision varies according to the plan and the quality of the material submitted. In cases where the cost of inspection of each piece is substantial, the reduction in number of pieces inspected may justify use of sequential sampling despite its greater complexity and higher administrative costs. On the other hand, where it is not practicable to hold the entire lot of parts while sampling and inspection are going on, it becomes necessary to set aside the full number of items that may need to be inspected before inspection even begins. In these circumstances single sampling may be preferable if the cost of selecting, unpacking, and handling parts is appreciable.

1.5 Sampling Risks and Parameters

1.5.1 Risks. When acceptance sampling is conducted, the real parties of interest are the producer (supplier or production department) and the consumer, i.e., the consumer, i.e., the

company which buys from the supplier or the department which is to use the product. Since sampling carries the risk of rejecting "good" lots and of accepting "bad" lots, with associated serious consequences, producers and consumers have gone far to standardize the concepts of what constitutes good and bad lots, and to standardize also the risk associated with sampling. These risks are stated in conjunction with one or more parameters, i.e., quality indices for the plan. These indices are as follows.

1.5.1.1 Producer's risk. The producer's risk α is the probability that a "good" lot will be rejected by the sampling plan. In some plans, this risk is fixed at 0.05; in other plans, it varies from about 0.01 to 0.10. The risk is stated in conjunction with a numerical definition of the maximum quality level that may pass through the plan, often called the acceptable quality level (AQL).

Acceptable Quality Level. As defined by MIL-STD-105D (1963), the acceptable quality level (AQL) is the maximum percent defective (or maximum number of defects per hundred units) that, for the purpose of sampling inspection, can be considered satisfactory as a process average. A sampling plan should have a low producer's risk for quality which is equal to or better than the AQL.

1.5.1.2 Consumer's risk. The consumer's risk β is the probability that a "bad" lot will be accepted by the sampling plan. The risk is stated in conjunction with a numerical definition of rejectable quality such as a lot tolerance percent defective (LTPD).

Lot Tolerance Percent Defective. The lot tolerance percent defective is the level of quality that is unsatisfactory and therefore, should be rejected by the sampling plan. A consumer's risk of 0.10 is common and LTPD has been defined as the lot quality for which the probability of acceptance is 0.10; i.e., only 10 percent of such lots will be accepted.

1.5.2 Operating Characteristic curve. The Operating Characteristic (OC) curve is a graph of lot fraction defective versus the probability that the sampling plan will accept the lot.

1.6 The Problem Defined

The discriminatory power of a sampling plan is revealed by its Operating Characteristic Curve, or OC curve as it is called *(Grant and Leavenworth, 1988)*. This curve shows how the probability of accepting a lot varies with the quality of the material offered for inspection. The concept is a simple one but requires careful exposition.

Dodge and Roming classify OC Curves into Type A and Type B OC curves. Type A curves give the probabilities of acceptance for various fractions defective as a function of the *lot* quality of finite lots. In principle, such curves should be computed by hypergeometric probabilities, the binomial or Poisson distributions often give satisfactory approximations.

Type B curves give the probabilities of acceptance of a lot as a function of *product* quality. Such curves are calculated as if the lot size were infinite. For Type B curves the binomial is exact and the Poisson often gives a satisfactory approximation.

A more accurate procedure for deriving an attributes single sampling plan is to use the *Larson binomial nomograph* (see Figure 1.3). The Larson nomograph is based on the binomial distribution and thus allows direct evaluation of Type B plans for fraction defective. It allows derivation and evaluation of plans for values of probability of acceptance not shown in the Cameron tables *(Schilling, 1982)*. It provides a reasonable and conservative approximation for the derivations of plans when the hypergeometric distribution should apply and the binomial approximation to the hypergeometric distribution is only approximate.

Nomograph of the Cumulative Binomial Distribution.

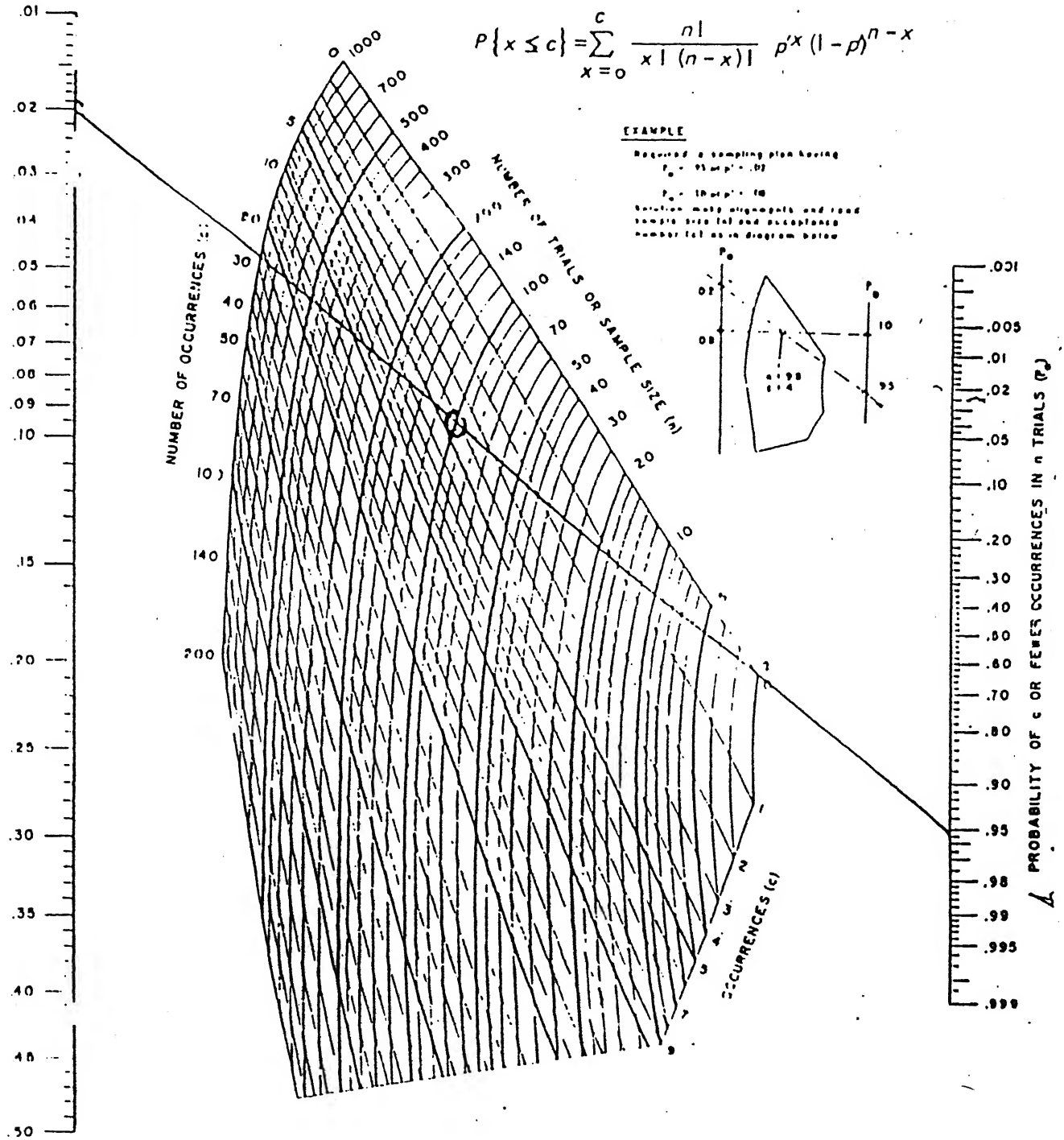


Figure 1.3

NOTE: If p' is less than 0.01, set kp' on the p'-scale and n/k on the n-scale, where $k = 0.01/p'$, rounded upward conveniently. Nomograph reproduced with permission from Harry R. Larson. "A Nomograph of the Cumulative Binomial Distribution," from the *Western Electric Engineer*, April 1965.

The Larson nomograph may also be used to derive single - sampling attributes plans as follows. Given $p_1(AQL)$, $1-\alpha$, $p_2(RQL)$, and β, plot p_1 and p_2 on the left scale for proportion defective shown as ''probability of occurrences on a single trial (p).'' Then plot $1-\alpha$ and β on the right scale for probability of acceptance shown as ''probability of c or fewer occurrences in n trials (p).'' With a straight edge, connect the points: p_1 with $1-\alpha$ and p_2 with β at the intersection of the lines, read the sample size n and the acceptance number c from the grid. One may note at this point that MIL-STD-105D sampling plans provide no explicit information about the risk protection they provide.

As evident from the methodology, described above, the sampling plan parameter values (e.g., n and c) require considerable analysis on part of the QC inspector. Artificial Neural Nets (ANNs) are acquiring increasing popularity in such tedious situations. In the context of acceptance sampling ANNs may be made to emulate the mathematical relations that govern the selection of a sampling plan by carrying out an off-line ''training'' on high speed computers. Once fully trained ANNs may provide instantaneous answers at the QC site also.

The present work has explored the possibility of such an endeavor with single sampling plans. Multi-layer perceptron (MLP) type ANNs trained on patterns containing risk, performance and sampling protocol data were used in this study.

CHAPTER 2
ARTIFICIAL NEURAL NETWORKS

2.1 Introduction.

Recent advances in neuroscience and in computer science have sparked a great deal of interest in neural network representations for problem solving. Neural networks are computing systems composed of a number of highly interconnected layers of simple neuron - like processing elements, which process information by their dynamic response to external inputs *(Kohonen, 1984; Rumelhart and McClelland, 1986)*. Computations are collectively performed by the entire network with knowledge represented as distributed patterns of activity over all processing elements. The collective operations result in a high degree of parallelism which enables the network to solve complex problems rapidly. The distributed representations leads to greater fault tolerance and to graceful degradation when problems are encountered beyond the range of experience of the network. In addition, neural networks can adjust dynamically to environmental changes, infer general rules from specific examples, and recognize invariances from complex high-dimensional data.

2.2 Fundamentals of Neural Computing.

This section summarizes the fundamentals of neural computing *(Quantrille and Liu, 1991)*. We discuss here the structure of the nodes that make up the ANN, as well as the nodes' interconnections. We discuss also the topology of ANNs, i.e., how the nodes are interconnected to develop a neural model. We then move on to training ANNs, and the different ways that ANNs can ''learn.''

2.2.1 Components of a Node

The foundation of an ANN is the artificial neuron, or node (sometimes called *neurode*). In most scientific and engineering applications, this is called a *processing element* (PE). Figure 2.1

shows the anatomy of a processing element.

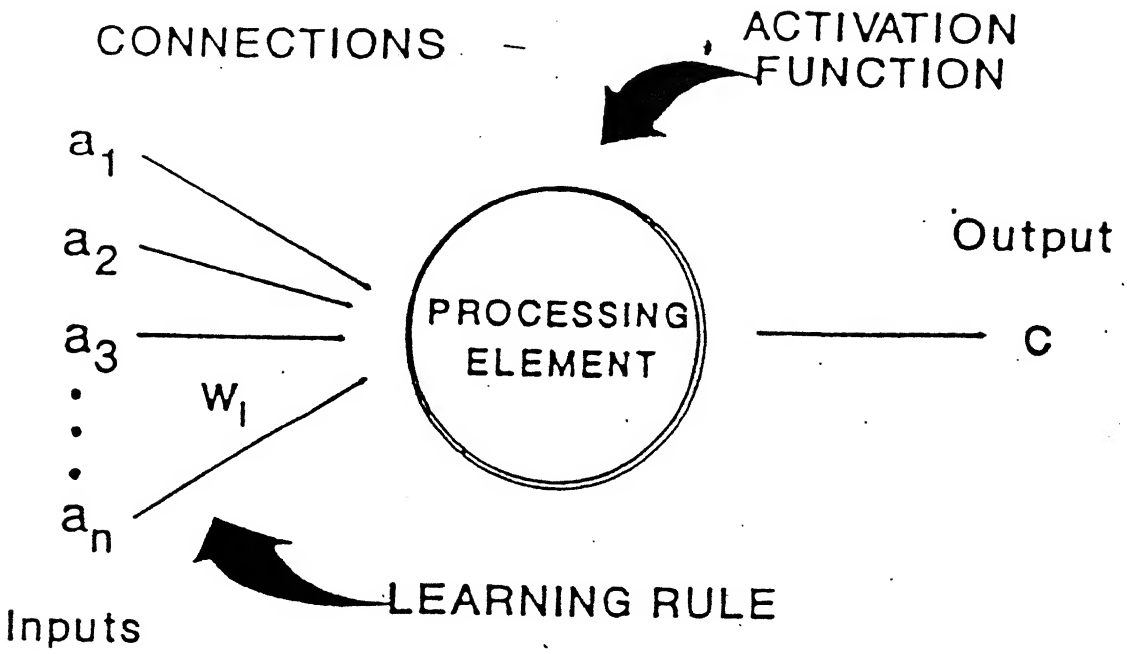


Fig. 2.1 Basic components of a neural network.

The PEs are the elements in the ANN where most calculations are performed.

1. Inputs and Outputs

The first element in the *j*th PE is an *input vector*, **a** with components a_1, a_2, ..., a_i, ..., a_n. The node manipulates these inputs, or activities, to give the output, b_j. This output can then form the part of the input for other PEs.

2. Weight Factors

Besides the component values of input vector **a**, some additional components of the PE also affect b_j. One such component is the *weight factor*, w_{ij}, for the *i*th input a_i corresponding to the *j*th node. Every input is multiplied by its weight factor.

For example, let us consider the processing element six. The first input into the element is a_1. Multiplying this input by the corresponding weight factor gives $a_1 w_6$. The PE uses this *weighted input* to perform further calculations.

Importantly, the weight factors can have an inhibitory or excitatory effect. If we adjust w_{ij} such that $a_i w_{ij}$ is positive (and preferably large), we tend to excite the PE. If $a_i w_{ij}$ is very small in magnitude relative to other signals, it (i.e., $a_i w_{ij}$) will have little or no effect on the node.

3. Internal Thresholds

The internal threshold for the *j*th PE, denoted T_j, controls activation of the node. The node calculates all its $a_i w_{ij}$'s, sums the terms together, and then calculates the total activation by subtracting the internal threshold value:

$$\text{Total Activation} = \sum_{i=1}^{n} (w_{ij} a_i) - T_j$$

If T_j is large and positive, the node has a high internal threshold, which inhibits node-firing. Conversely, if T_j is zero (or negative, in some cases), the node has a low internal threshold, and that low value of T_j excites node-firing.

Some, but not necessarily all, PEs may have an internal threshold level. If no internal threshold is specified, we assume T_j to be zero.

4. Functional Forms

The PE performs calculations based on its input. It takes the dot product of vector **a** with vector $W = [W_{1j}, W_{2j}, \ldots, W_{nj}]^T$, subtracts the threshold T_j, and passes this result to a functional form, f(). Thus, the node calculation is:

$$f(W * a - T_j) = f(\sum_{i=1}^{n} (W_{ij} a_i) - T_j)$$

This calculation, then, is a function of the *difference*, or *error* between the input function and the internal threshold.

What functional form do we choose for f()? We could choose whatever we want —— square root, $\prod$ (product), log, e^x and so on.

Mathematicians and computer scientists have found that the *sigmoid* (S-shaped) function is particular advantageous. A typical sigmoid function has the form

$$f(x) = \frac{1}{1 + e^{-\beta x}}$$

which is shown in Figure 2.2.

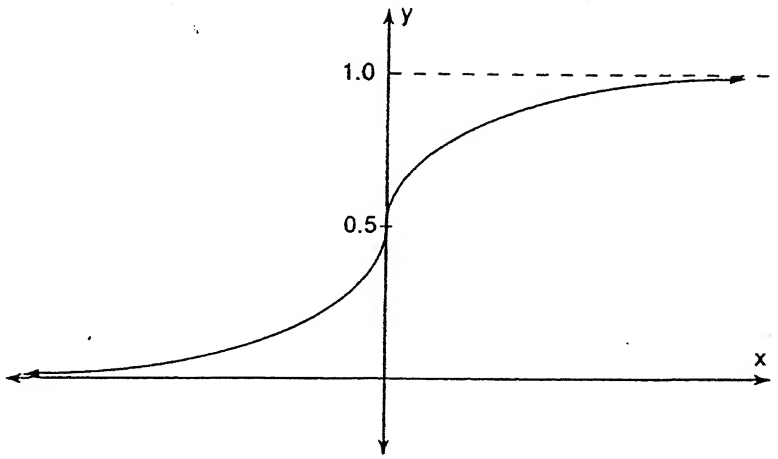


Fig.2.2 A sigmoid (S-shaped) function.

This function is monotonically increasing, with limiting values of 0 (at $x = -\infty$) and 1(at $x = +\infty$). Because of these limiting values, sigmoid functions are also called *threshold functions*. At very low input values, the threshold-function output is zero. At very high input values, the output value is 1.0. All sigmoid functions have upper and lower limiting values.

Another sigmoid function used is the hyperbolic tangent:

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

with limiting values of -1 and +1. A third common sigmoid function is the ratio of squares:

$$f(x) = \begin{cases} \dfrac{x^2}{1 + x^2}, & \text{if } x > 0 \\ 0 \text{ if } x <= 0. \end{cases}$$

with limiting values of 0 and 1.

The sigmoid (S-shaped) functions when used in general, leads

to fairly well-behaved ANNs. The inhibitory and excitatory effects of the weight factors are straightforward when we use sigmoid functions (i.e., $w_{ij} < 0$ is inhibitory, and $w_{ij} > 0$ is excitatory). Sigmoid functions are continuous and monotonic, and are well-behaved even as x approaches $\pm\infty$. Because they are monotonic, they also provide for more efficient *training*. We frequently move down the slope of the curve in training, and the sigmoid functions have slopes that are well-behaved as function of x (this topic will be discussed later in the section on gradient-descent learning).

5. Summary of Processing-Element Anatomy

The fundamental building block of an ANN is the processing element (PE) which has associated with it an n-dimensional input vector, a, an internal threshold value, T_j, and n weight factors (w_{1j}, ..., w_{ij}, ..., w_{nj}) which multiply all inputs. If the weighted input is large enough, the node becomes active and performs a calculation based on the difference between the weighted input value and the internal threshold value. Typically, a sigmoid function is used for f(x), since it is a threshold function, is well-behaved, and provides for more rapid training.

2.2.2 Topology of an Artificial Neural Network

The *topology* of an ANN refers to how its PEs are interconnected. Figure 2.3 shows a very common topology.

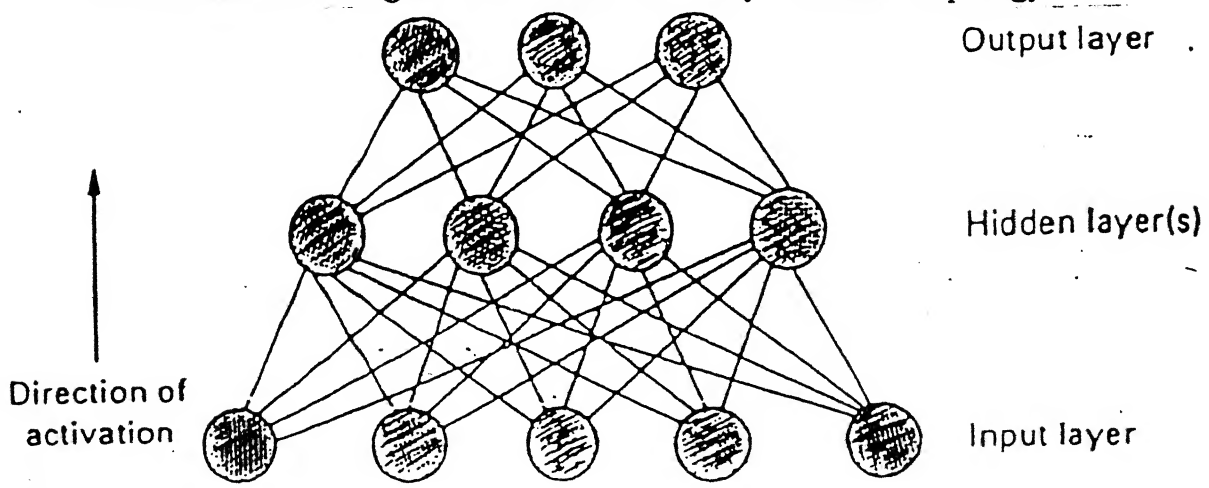


Fig. 2.3

This network has three layers, one hidden, and each node's output feeds into all nodes in the subsequent layer. We form these topologies, or architectures, by organizing the PEs into layers, connecting them, and ''weighing'' the interconnections.

1. Inhibitory or Excitatory Connections

As mentioned, connection can either inhibit or excite the node. If the weight is positive, it will excite the node, increasing the activation of the PE. If the signal is negative, it will inhibit the node, decreasing the activation of the PE. If the signal is highly inhibitory, it may lower the input below the threshold level and shut the node down.

2. Connection Options

Generally, when we first build an ANN, we *pre-specify the topology*. That is, we specify the interconnections, but leave the numerical values of the weights up to the training phase. There are two options available to us viz.,*feedforward connection,* and *feedback connection*. There is limited availability of rigorous mathematical theory for training ANNs. Most approaches are based on feedforward networks, since they are the most applicable once in science and engineering, the least complicated, and straightforward to implement.

An ANN having no intermediate layer forms the fundamental *perceptron.* The perceptron was conceived by Rosenblatt in 1958 *(Rosenblatt, 1959)*. Its input is an n-dimensional vector. The perceptron computes a weighted sum of the n components of the input vector and then adds a *bias value* to this sum. The result is then passed through a nonlinearity, defined as:

$$f_{hl}(y) = \begin{cases} 1, & y>0 \\ 0, & y<=0 \end{cases}$$

2.2.3 Introduction to Learning and Training.

To train ANNs, we adjust the weight factors until the calculated output pattern (response) based on the given input

matches the desired cause-and-effect relations. *Learning* is the actual process of adjusting weight factors based on trial-and-error. With ANNs, this task is challenging in several respects.

1. Stability and Convergence

The training phase needs to produce an ANN that is both *stable* and *convergent*. A globally stable ANN maps any set of inputs to a fixed output. Stability guarantees a result, but it does not necessarily guarantee an accurate result.

A *convergent* ANN produces *accurate* input-output relations. Convergence, then, is related to the accuracy of the ANN. The magnitude of error between real-world results and those predicted by the ANN is a direct measurement of ANN's convergence.

2. Types of Learning

There are many different approaches to training ANNs. Simpson (1989), in his book *Artificial Neural Systems*, classifies a number of approaches. Most approaches fall into one of two groups:

* *Supervised learning* —— an external teacher controls the learning and incorporates global information.

* *Unsupervised learning* —— no external teacher is used and the ANN relies upon both internal control and local information. Frequently, the ANN develops its own models without additional input information.

We discuss some specific learning procedures below.

Error-Correction Learning —— This is the most common type of learning used in developing ANNs today. It is a form of supervised learning, where we adjust weights in proportion to the output error vector, ϵ. This output error vector has n components, where n is the number of nodes in the output layer. We denote the *n*th

component of the vector as the vector as ϵ_n, and thus obtain an error component for each output from the ANN.

We begin error-correction learning by defining the output error from a single node on the output layer as:

$$\epsilon_n = d_n - b_n$$

where ϵ_n is the output error, d_n is the desired output, and b_n *is* the calculated output, for the *nth* node in the *output layer only*. We then calculate the total squared error of the output layer, E, as:

$$E = \sum_n \epsilon_n^2 = \sum_n (d_n - b_n)^2$$

Knowing E, we can calculate the change in the weight factor (the effect of learning) for the *i*th connection to the *j*th node, Δw_{ij}:

$$\Delta w_{ij} = \beta_j a_i E$$

where β_j is a linear proportionality constant for node j (typically, $0 < \beta_j << 1$), and a_i is the *i*th input to node j.

Reinforcement Learning —— This type of supervised learning is closely related to error-correction learning. In error-correction learning, we calculate a *vector* of error values, ϵ. Thus, n different error values represent the total performance of the ANN. In contrast, reinforcement learning uses only one *scalar* error value to represent to total performance of the ANN. Since we have only one value to reckon with, reinforcement learning is generally easier and simpler than the classical error-correction learning. Reinforced learning is ''selectively supervised,'' and requires less information, possibly at infrequent intervals.

Stochastic Learning ——— This procedure utilizes statistics, probability, and/or random processes to adjust connection weights. We accept a random weight change if it reduces the error vector,

ϵ. If the change increases ϵ, we generally reject the change. However, we may accept this change if, according to a specifically encoded probability analysis, it has a better-than-average probability of moving us to a global minimum in error. Random acceptance of seemingly poorer weights allows stochastic learning to escape local minima and move to the global minimum.

Hardwired ANNs have all connections and weights predetermined (hardwired). They have a *priori* information in speech recognition, language processing, vision, and robotics.

Hebbian Learning (named after *Donald Hebb, 1949*) adjusts weights based on a correlation between the two nodes that the weight is associated with. The simplest form of Hebbian learning uses direct proportionality. With node a_i in one layer connected to node b_j in another layer, we adjust the weight w_{ij} to reflect learning according to the equation

$$w_{ij} = w_{ij,old} + \beta_j a_i b_j$$

where β_j is a learning rate constant for node j, and $0<\beta_j<1$.

CHAPTER 3
NEURAL NETWORKS AT WORK

3.1 Introduction.

The multilayered perceptron neural networks trained by BP produce hetero and auto pattern associators which can be applied to a wide range of problems especially to process monitoring and control in industrial settings. The two broad categories where the applications of neural networks can fall are defined as recognition problems which are auto associative where the network recognizes noise and the generalization problems which can be either classification or prediction *(Smith and Dagli, 1991)*.

Having been used experimentally now for two decades, neural networks are reputedly a solution in search of a problem. More recently, though, they began moving into practical applications, and this trend can only accelerate now that specialized hardware is available to speed product development.

Hundreds of actual applications use these networks, often without public acknowledgement, to preserve a competitive advantage. products known to be based on them include facsimile optical character recognition (OCR) systems from Calera Recognition Systems Inc., Santa Clara, CA; electrocardiograph and pap-smear systems from Neuromedical Systems Inc., Suffern, NY; process control from Pavilion Technologies Inc., Austin, TX; and financial systems from HNC Inc., San Diego, CA, and other companies. Military uses such as target recognition and flight control have also been reported, as well as communications applications like adaptive echo cancellation.

Two representative applications *(Hammerstrom, 1993)* will be described here in enough detail to show how ANNs work. Some perform a task for which a solution already exists, but which is made more competitive, reliable, or profitable by the neural network. Others solve hitherto intractable problems and so generate ground-breaking products. What the applications share is

the need to cope with noise, imprecision, and complexity; that is, they do ''ordinary'' things in the real world.

3.2 Benefits of using Neural Networks.

Neural networks are valuable on several counts. First, they are adaptive: they can take data and learn from it. Thus they infer solutions from the data presented to them, often capturing quite subtle relationships.

This ability differs radically from standard software techniques because it does not depend on the programmer's prior knowledge of rules. Neural networks can reduce development time by learning underlying relationships even if they are difficult to find and describe. They can also solve problems that lack existing solutions.

Second, neural networks can generalize: they can correctly process data that only broadly resembles the data they were trained on originally. Similarly, they can handle imperfect or incomplete data, providing a measure of fault tolerance. Generalization is useful in practical applications because real-world data is noisy.

Third, the networks are nonlinear, in that they can capture complex interactions among the input variables in a system. In a linear system, changing a single input produces a proportional change in the output, and the input's effect depends only on its own value. In a nonlinear system, the effect depends on the values of other inputs, and the relationship is a higher-order function.

Systems in the real world are often nonlinear. In a country's economy, for instance, prices, interest rates, employment level, and other factors react with each other. The effect of a price change might depend on, for example, the interest rate, so the same change leads to different contexts. Neural network offer a practical approach to such complex systems.

Fourth, neural networks are highly parallel: their numerous identical, independent operations can be executed simultaneously. Parallel hardware can execute them hundreds or thousands of times faster than conventional microprocessors and digital signal processors. Now that special-purpose parallel computers and chips have become available for neural network implementations, even large networks can achieve real-time speeds, and even everyday products can employ an embedded network. This increase in speed and economy makes many applications practical for the first time, encouraging further deployment.

3.3 Types of Applications.

A pattern recognition application using a neural network may be categorized in terms of how it employs the network. The two largest categories cover the use of a neural network for *classification or function estimation.* Other categories include data compression, feature extraction, and statistical clustering. Each basic capability enables many specific applications. A neural network classifier combined with a camera, for instance, may grade potatoes, read zip codes, or match fingerprints to a database. Combined with a microphone, it may diagnose engine trouble, identify underwater sounds, or generate signals that cancel vibrations.

3.3.1 Classification Application:
ANN as a Magnetic-ink Character Recognizer(MICR).

Magnetic-ink character recognition numbers are the digits at the bottom of checks that identify an account number. They were designed to be read magnetically, so they are printed in an ANSI-standard font having only 14 characters. Unlike multi-font kanji recognition, MICR recognition is a tightly constrained problem.

Although banks have long had equipment for reading MICR numbers, banks are not the only place where MICR data is useful. Retail businesses also accept billions of checks, absorbing large

losses from bad checks in the process. To reduce this loss, many retailers now subscribe to electronic check approval services, which list high-risk accounts by MICR number.

MICR readers are available for retail use, but read only about 80 percent of consumer checks on the first pass. They are also expensive, so most retailers still key in the MICR line by hand. Manual entry is slow and prone to error, with a digit mis-keyed in about one in 10 transactions. Plainly there is a need for an accurate, low-cost device similar to the magnetic-stripe readers for credit cards.

Its design must satisfy several constraints to be competitive: it must be accurate, small, economical, and easy to use. In 1992, VeriFone, a supplier of transaction automation systems, announced a magnetic MICR reader aimed at retailers. According to VeriFone, its Onyx check reader meets its design goals specifically because it contains an embedded neural network. For example, the company claims 99.6 percent accuracy even with checks that are folded, crumpled, or overwritten.

The heart of VeriFone's system is a custom chip designed and manufactured by Synaptics, founded in 1986 by Carver Mead and Federico Faggin. This chip combines a retina-like optical sensor with a neural network designed to classify MICR characters. Using analog VLSI for high density and high speed, the I-1000 chip classifies 1000 MICR characters per second while consuming only 10mW. This small, inexpensive, low-power CMOS design helped VeriFone keep the system compact and low-cost, as needed for a point-of-sale device.

How does Synaptics' chip work? Briefly, an infrared light-emitting diode (LED) illuminates the check so that a lens can project an image onto the chip's ''retina,'' which captures an image at 400 pixels per character. This image serves as the input to two neural networks that work together to classify the digits. Keeping the sensor and classifier on one chip reduces recognition time and simplifies manufacturing.

The classification section consists of two neural networks, one for locating characters and the other for identifying them. The location network detects the vertical position of a character within its legal range. Its inputs are continuous-time data from the on-chip retina, and its output is the most likely of about 40 possible vertical positions. After locating a character, the system passes the image to the second neural network.

The identification network has 14 output nodes, one for each MICR character. All 14 nodes see all inputs simultaneously;when the node with the largest output exceeds a predetermined threshold, the network has identified a character. The system then passes the character code to a single-chip microcontroller used for control and communications.

The use of a neural network offers several benefits here. For one, the network ''sees'' each character as a complete image, so the check reader is largely immune to folding, smudging, and other defects. for another, the network looks for characters continuously, so its accuracy is largely independent of how quickly the check passes through the reader. This eliminates the need for a motor drive or position wheel, simplifying the system design. It also improves ease of use, since the user can slide checks through the reader by hand without a prescribed motion.

VeriFone's system shows that using a dedicated, low-cost chip to embed a neural network in a mass-produced product can make it more competitive, fostering acceptance in a new market. Such custom implementations will flourish everyday products start to adopt pattern recognition technology.

3.3.2 Function Estimation.

In function estimation, another major branch of neural network applications, the neural network is useful because it acts as a model of a real-world system or function. The model then stands for the system it represents, typically to predict or

control it.

Applications of this kind have vast potential. Nearly any industrial process is a candidate, including such diverse tasks as molding plastics, forecasting power loads, and canceling noise. Many statistical applications are also candidates, including rating credit applications, detecting fraud, and even predicting the weather.

As mentioned earlier, neural networks include adaptivity and nonlinearity among their useful properties. Both are well suited to many function estimation tasks in the real world. Adaptivity, for example, means that a neural network can model a function even if the equation describing it is unknown -- the only prerequisite is a representative sample of the function's behavior. Neural networks therefore offer the opportunity to manage systems from their observed behavior, not from a theoretical understanding.This ability to learn from experience is invaluable in the real world.

Their nonlinearity is also invaluable, since real-world systems often have known and unknown nonlinear relationships. The many unknown linear methods applied to them do not necessarily yield ideal solutions, but can at least be dealt with mathematically. Linear regression, for example, has known limitations for nonlinear problems -- but one uses the tools to hand. Here, neural networks offer a chance to match a problem in complexity.

One example of neural networks as function estimators is a financial forecasting application developed at University College London, to predict the bond market from its historical behavior. The other is a process control application by Fujitsu, which monitors a continuous casting operation to reduce loss from a molding failure. In both cases, the neural network models a multivariate process, attaining a much better result than the conventional method.

Financial Forecasting.

Financial forecasters now make money the new-fashioned way: they learn it. That is, they train a neural network to mimic a market, then use its predictions to guide investment.

In fact, many well-known companies are quietly using neural networks for various financial tasks. For the most part, they say very little about their methods and results. That is to be expected, since neural networks have demonstrated an outstanding talent for turning a real money.

In 1992, the London Business School and University College London established their NeuroForecasting Centre to produce neural-network-based systems for financial decision making. The Centre's initial systems focus on foreign exchange, stock and global capital markets. According to the Centre's director, A.N.Refenes, prototype systems have done far better than conventional statistical approaches. In foreign exchange, for instance, a system using a neural network to select trading strategies earned an average annual profit of 18 percent on a US $1 million position.

The local neural networks are trained with historical data about the bond market in the country they represent. Each uses as inputs four to eight parameters (such as interest rates, oil prices, and the ratio of precious to nonprecious metals) on a month-to-month basis between 1974 and 1988. The target output is the bond return for the next month.

During training, each network passes through the historical data repeatedly to learn how the input parameters affect each market. Ten percent of the data is held back for testing to determine when to end training.

A system like the one described went live in October 1992 with a $10 million investment. Its average performance during November and December was 2.8 percent above a benchmark.

CHAPTER 4
THE BACKPROPAGATION ALGORITHM

4.1 The Simple Backpropagation Algorithm.

The core of the learning logic for numerous ANNs, ''backpropagation learning'' attempts to properly map given inputs with desired outputs by minimizing an error function, typically the sum-of-squares error. The component of the output error vector ϵ_n which forms the *n*th node on the output layer is defined as $\epsilon_n = d_n - c_n$, where d_n is the desired output value and c_n is the calculated value. The total (squared) error function, *E*, then is

$$E = \sum \epsilon_n^2 = \sum (d_n - c_n)^2,$$

We adjust both interconnection weights, v_{ij}'s and w_{jk}'s, shown in Figure 4.1 to minimize *E*.

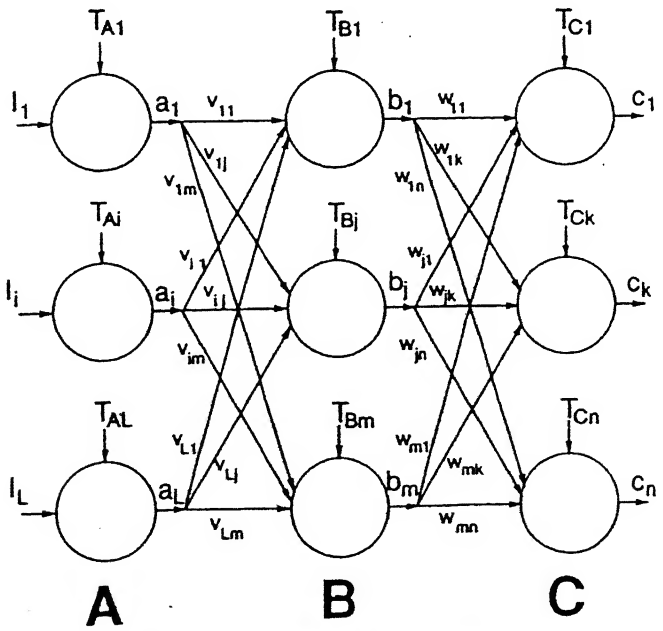


Fig 4.1 A three-level perceptron ANN.

Below is the step-by-step adjustment procedure known as the *vanilla backpropagation algorithm* due to *Simpson (1989)*. To use this algorithm, we must have sets of data that map vector I_L with the output c_n.

Step 1: Randomly specify numerical values for all weight factors

(v_{ij}'s and W_{jk}'s) within the interval [-1, +1]. Likewise, assign internal threshold values (T_{Ai}, T_{Bj}, T_{Ck}) to every node, also between +1 and -1. Note that i = 1,2, ..., L, where L is the number of nodes in layer A; j = 1,2, ..., m, where m is the number of nodes in layer B; and k = 1,2, ..., n, where n is the number of nodes in layer C.

Step 2: Introduce the input I_i into the ANN. Calculate all outputs from the first layer, using the standard sigmoid function introduced previously as

$$x_i = I_i - T_{Ai}$$

$$a_i = f(x_i) = \frac{1}{1 + e^{-x_1}}$$

Here, I_i is the input into the *i*th node on the input (i.e.,A) layer, T_{Ai} is internal threshold for the node, and a_i is the output from the node.

Step 3: Given the output from later A, calculate the output from layer B, using the equation

$$b_j = f\left(\sum_{i=1}^{L} (v_{ij}\, a_i) - T_{Bj}\right)$$

where f() is the same sigmoid function.

Step 4: Given the output from layer B, calculate the output from layer C, using the equation:

$$C_k = f\left(\sum_{j=1}^{m} (w_{jk} b_j) - T_{ck}\right)$$

where f() is the same sigmoid function.

Step 5: Now *backpropagate* through the network, starting at the output and moving backward toward the input. Calculate the *k*th component of the output error, ϵ_k, for each node in layer C, according to the equation

$$\epsilon_k = c_k(1-c_k)(d_k-c_k)$$

where d_k is the desired result and c_k is the actual result.

Step 6: Continue backpropagation, moving to layer B. Calculate the *j*th component of the error vector, e_j, of layer B relative to each ϵ_k, using the equation:

$$e_j = b_j(1-b_j)\left(\sum_{k=1}^{n} (w_{jk}\epsilon_k)\right)$$

Step 7: Adjust weights, calculating the new w_{jk} ($w_{jk,new}$) as:

$$w_{jk,new} = w_{jk,old} + \beta_c b_j \epsilon_k$$

for j = 1 to m and k = 1 to n. The term β_c is a positive constant controlling the learning rate in layer C.

Step 8: Adjust the threshold T_{ck} (k = 1 to n) in layer C, according to the equation

$$T_{ck,new} = T_{ck,old} + \beta_c \epsilon_k$$

Step 9: Adjust weights v_{ij}, according to the equation

$$v_{ij,new} = v_{ij,old} + \beta_B a_i e_j$$

for i = 1 to L and j = 1 to m. The term β_B is a positive constant controlling the learning rate in layer B.

Step 10: Adjust the threshold T_{Bj} (j = 1 to m) in layer B, according to the equation

$$T_{Bj,new} = T_{Bj,old} + \beta_B e_j$$

Step 11: Repeat steps 2-10 until the squared error, E, or the output error vector, ϵ, is zero or sufficiently small.

This algorithm requires a few comments. We do *not* adjust the internal threshold values for layer A. Therefore, depending on

the problem being solved, we may wish to set all T_{Ai}'s equal to zero. Also, in this network, the sigmoid function restricts the output to values between zero and one. This restriction can cause some problems when the ANN is used for empirical modeling. If we desire values outside the [0,1] interval, some type of ''normalized output value is required.

4.1.1 Sigmoid Threshold Functions in Backpropagation

The ANN just described has a potential problem: *the presence of the internal threshold values,* T_{Ai}, T_{Bj}, and T_{Ck}. As these threshold values are adjusted during backpropagation, node-activation levels (as determined by weights from the previous time step) can suddenly be excessively low or high, depending on the new threshold value. These sudden changes in node activation may induce discontinuities in the network. The discontinuities may, in turn, lead to *poor network stability.* During training, the network may be prone to *oscillations.* In addition, mathematicians have proven that such networks are *not able to simulate* all types of mathematical functions, such as the ''exclusive-or'' gate used in logic and Boolean algebra.

Because of these difficulties, many ANNs *do not use internal threshold values* (i.e., the T_{Ai}'s, T_{Bj}'s, T_{Ck}'s are all set to zero). Instead, the sigmoid function itself acts as a ''gradual threshold,'' since it has a limiting value of zero at low activation and unity at high activation. Figure 4.2 shows a network with these properties. The sigmoid functions deactivate the node, and hence, they are called *sigmoid threshold functions.*

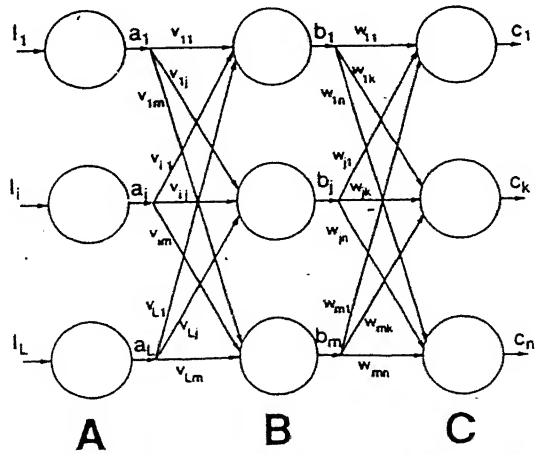


Fig 4-2 An ANN with sigmoid threshold functions. All T_{Ai}'s, T_{Bj}'s, and T_{Ck}'s are set to zero.

4.2 Generalized Delta-Rule (GDR) Algorithm

Today, however, with the growing number of more sophisticated training algorithms, many of these algorithms use both the internal threshold and sigmoid threshold functions. An example is GDR algorithm, a gradient-descent learning technique that we describe below.

1. *The Generalized Delta-Rule* (GDR) *Algorithm*

One difficulty with backpropagation algorithms is the extensive time required to train the network. Depending on the size of the ANN, training alone can take hours or even days on a mainframe computer. Researchers have investigated many different training procedures in an attempt to speed up the learning process. One technique that has been frequently applied is gradient-descent learning.

The *Generalized Delta-Rule* (GDR) is an iterative gradient-descent method that minimizes the squared error. It is related to the vanilla backpropagation algorithm, but has some differences. First, the GDR uses a technique known as *momentum* to speed up the training. Momentum is an extra weight added onto the weight factors when they are adjusted. By accelerating the change in the weight factors, we improve the training rate.

Another difference between the GDR and the vanilla bakpropagation algorithm is the presence of a *bias function* in place of internal threshold values. The internal threshold (T_{1i}, T_{2j}, and T_{3k}) become a bias function when we *add* (rather than subtract as in the vanilla backpropagation) a fixed number to the nodal summation. In addition, when serving as a bias function, the values of T_{1i}, T_{2j}, and T_{3k} are *not* changed or updated as training progresses. In GDR, we set $T_{1i} = 0$, and $T_{2j} = T_{3k} = 1$, and these values remain unchanged throughout the entire life of the ANN.

By using momentum coupled with a bias function, the GDR algorithm becomes more efficient than the vanilla backpropagation

algorithm. It goes as follows:

Step 1: Randomly assign values between 0 and 1 to weights v_{ij} and w_{jk}. For GDR, the internal threshold values *must* be assigned as follows: all input-layer thresholds must equal zero, i.e., T(1,i) = 0; all hidden- and output- layer thresholds must equal one, i.e., T(2,j) = T(3,k) = 1.

Step 2: Introduce the input vector I_i into the ANN, and calculate the output from the first layer according to the equations

$$x_i = I_i - T_{1i} = I_i - 0 = I_i$$

$$a_i = \frac{1}{1 + e^{-x_1}}$$

Step 3: Knowing the output from the first layer, calculate outputs from the second layer, using the equation

$$b_j = f\left(\sum_{i=1}^{L} (v_{ij}\, a_i) + T_{2j}\right)$$

where f() is the sigmoid function. Note that $T_{2j} = 1$ in this algorithm, and we are *adding* it to the summation (rather than *subtracting* it as we did in the vanilla backpropagation algorithm). When used in this mode, T_{2j} acts as a bias function.

Step 5: Continue steps 1-4 for M number of training patterns presented to the input layer. Calculate the total-squared error, E, according to the following equation

$$E = \sum_{m=1}^{M} \sum_{k=1}^{n} (d_k^m - c_k^m)^2$$

where M is the number of training patterns presented to the input layer, n is the number of PEs on the *k*th PE in the *m*th training pattern, and c_k^m is the actual output value from the

*k*th PE in the *m*th training pattern.

Step 6: Knowing the *m*th pattern, calculate δ^m_{3k}, the gradient-descent term for the *k*th PE in the *output layer* (layer 3) for training pattern m. Use the following equation:

$$\delta^m_{3k} = (d^m_k - c^m_k) \frac{\partial f}{\partial x_k}$$

where f is the sigmoid function:

$$f(x_k) = \frac{1}{1 + e^{-x_k}}$$

The partial derivative of the sigmoid function is:

$$\frac{\partial f}{\partial x_k} = \frac{e^{-x_k}}{(1 + e^{-x_k})^2}$$

Note that x_k is the sum of the weighted inputs to the *k*th node on the output layer i.e., for the *m*th training session:

$$x^m_k = \sum_j w^m_{jk} b^m_j + T^m_{3k}$$

where, for training pattern m, b^m_j is the output value of the *j*th element in the hidden layer and T_{3k} is a threshold value on the output layer.

Step 7: Again knowing the *m*th pattern, calculate δ^m_{2j}, the gradient-descent term for the *j*th PE on the *hidden layer* (layer 2). Use the equation:

$$\delta^m_{2j} = (\sum_k \delta^m_{3k} w^m_{jk}) \frac{\partial f}{\partial x_j}$$

where the subscript k denotes a node in the output layer. Recall that x_j is defined by:

$$x_j^m = \sum_i v_{ij}^m a_i^m + T_{2j}^m$$

and the partial derivative of the sigmoid function, again, is:

$$\frac{\partial f}{\partial x_j} = \frac{e^{-x_j}}{(1 + e^{-x_j})^2}$$

Step 8: Knowing δ_{2j}^m for the hidden layer and δ_{3k}^m for the output layer, calculate the weight changes using the equations:

$$\Delta v_{ij}^m = \eta \, \delta_{2j}^m \, a_i^m + \alpha \, \Delta v_{ij}^{m-1}$$
$$\Delta w_{jk}^m = \eta \, \delta_{3k}^m \, b_j^m + \alpha \, \Delta w_{jk}^{m-1}$$

where η is the *learning rate,* and α is a coefficient of *momentum*. As mentioned, momentum is simply an added weight used to speed up the training rate. The coefficient of momentum, α, is usually restricted such that $0 < \alpha < 1$. Thus, the momentum terms, $\alpha\Delta w_{jk}^{m-1}$ and $\alpha\Delta v_{ij}^{m-1}$, are fractional values of the weight change from the previous iteration.

Step 9: Knowing the weight changes, update the weights according to the equations:

$$w_{jk}^m = w_{jk}^{m-1} + \Delta w_{jk}^m$$
$$v_{ij}^m = v_{ij}^{m-1} + \Delta v_{ij}^m$$

where v_{ij}^m is the connection weight between the *i*th element in the input layer and *j*th element in the hidden layer, w_{jk}^m is the connection weight between the *j*th element in the hidden layer and *k*th element in the output layer, both for the *m*th training iteration. Repeat steps 2-9 for all training until the squared error is zero or sufficiently low.

From these steps, we see that the *Generalized Delta Rule* computes the output error, then generates new weight values based

on the gradient of the sigmoid function and that error. It updates the output node first, and propagates back to the next layers, until it reaches the input layer. It also uses momentum and a bias function, which are two distinguishing features between the GDR and the vanilla backpropagation algorithms.

4.3 Backpropagation with the Generalised Delta Rule

Summarily the algorithm proceeds as follows to optimize the connection weights: To begin learning the network is initialized with small random weights on each arc. A training pattern, m is taken and the input vector I_L is propagated through the network to get the predicted output, c_n. A gradient δ^m_{2j} and δ^m_{3k} in the space of network weights is then calculated using the GDR. Knowing the gradients weight changes Δv^m_{ij} and Δw^m_{jk} are evaluated followed by weights updation w^m_{jk} and v^m_{ij} according to the equations:

$$w^m_{jk} = w^{m-1}_{jk} + \eta\delta^m_{3k} b^m_j + \alpha\Delta w^{m-1}_{jk}$$
$$v^m_{ij} = v^{m-1}_{ij} + \eta\delta^m_{2j} a^m_i + \alpha\Delta v^{m-1}_{ij}$$

In the above expressions appear two constants, η called the learning rate which is equivalent to a step size, and α which adds a momentum term to keep the direction of descent from changing too rapidly from step to step. After the weights are updated, a new training example is fed and the procedure repeated until satisfactory reduction of the objective function is achieved.

CHAPTER 5
THE MULTI-LAYERED PERCEPTRON: Issues and Limitations

5.1 Introduction.

The Multi-Layered Perceptron (MLP) is capable of approximating arbitrary nonlinear mappings, and given a set of examples, the backpropagation algorithm can be used to learn the mapping at the example points. However, there are a number of practical concerns. The first is the matter of choosing the network's size. The second is the time complexity of learning. That is, we may ask if it is possible to learn the desired mapping in a reasonable amount of time. Finally, we are concerned with the ability of our network to generalize; that is, its ability to produce accurate results on new samples outside the training set.

5.2 Choosing the Network Size.

Theoretical studies suggest that the MLP is capable of forming arbitrarily close approximations to any continuous nonlinear mapping. However, this is true only when the size of the network is allowed to grow arbitrarily large *(Cybenko, 1989)*. In general, it is not known what (finite) size network would work best for a given mapping problem. Further, it is not likely that this issue will be resolved in the general case, since each problem will likely demand different capabilities from the network. Choosing the proper network size is important for two reasons. If the network is too small, it will not be capable of forming a good model of the problem. On the other hand, if the network is too big then it may be *too capable (Baum and Haussler, 1989)*. That is, it may be able to implement *numerous solutions* that are consistent with the training data, but most of these are likely to be poor approximations to the actual problem. In this case, the solution learned during any given training session is likely to be a poor approximation to the actual problem.

Ultimately, we would like to find a network whose size best matches the capability of the network to the structure of

underlying problem; or, since the data is sometimes not sufficient to describe all of the intricacies underlying problem, we would like a network whose size best captures the structure of the data. With some specific knowledge about the structure of the problem (or data), and a fundamental understanding of how the MLP might go about implementing this structure, one may sometimes form a good estimate of the proper network size.

With little or no prior knowledge of the problem, however, one must determine the network size by trial and error. A methodical procedure is recommended. One approach is to start with the smallest possible network and gradually increase the size until the performance begins to level off. In this approach, each size network is trained independently. A closely related approach is to ''grow a network.'' The idea here is to start with one node and create additional nodes as they are needed.

Another possibility is to start with a large network and then apply a pruning technique that destroys weights and/or nodes which end up contributing little or nothing to the solution *(Cun et al., 1990)*. With this approach one must have some idea of what size network constitutes a ''large'' network; that is, ''what size network is probably too big for the problem?''

The following guidelines are useful in placing an upper bound on the network size. For a fully connected MLP network, no more than three layers are typically used (*i.e., not more than two hidden layers in any case*), and in most cases only two (*one hidden layer*). Numerous bounds exist on the number of hidden layer nodes needed in 2-layer networks. For example, it has been shown that an upper bound on the number of hidden layer nodes needed for the MLP to implement the training data exactly is on the order of P, the number of training samples *(Huang and Huang, 1991)*. This suggests that one should never use more hidden layer nodes than training samples. Actually, the number of hidden layer nodes should almost always be much less than the number of training samples, otherwise the network simply ''memorizes'' the training samples, resulting in poor generalization.

5.3 Complexity of Learning.

Even if one is able to determine the optimal network size, it turns out that finding the correct weights for a network is an inherently difficult problem. The problem of finding a set of weights for a fixed-size network which performs the desired mapping exactly for some training set is known as the *loading problem*. Recently it has been shown that the loading problem is NP-complete *(Blum et al., 1988; Judd, 1990)*. This suggests that if we have a very large problem, e.g., if the dimension of the input space is very large, then it is unlikely that we will be able to determine if a weight solution exists in a reasonable amount of time.

On the other hand, learning algorithms like backpropagation are based on a gradient search, which is a greedy algorithm that seeks out a local minimum and thus may not yield mapping. Gradient search algorithms usually don't take exponential time to run (if a suitable stopping criteria is used). However, it is well known that finding local weight solutions using backpropagation is extraordinarily slow.

It is dangerous to increase the learning rate to compensate for the sluggishness because the algorithm may then exhibit instabilities. Attempts to speed learning include variations on simple gradient search *(Jacobs, 1988; Plaut et al., 1986; Rumelhart, Hinton et al., 1986)*, line search methods *(Hush and Salas, 1988)*, and second-order methods *(Becker et al., 1988; Watrous,1987)*. Although most of these have been somewhat successful, they usually introduce additional parameters which are difficult to determine, must be varied from one problem to the next, and if not chosen properly can actually slow the rate of convergence.

5.4 Generalization.

Generalization is a measure of how well the network performs on the actual problem once the training is complete. It is usually

tested by evaluating the performance of the network on new data outside the training set. Generalization is most heavily influenced by three parameters: the number of data samples (and how well they represent the problem at hand), the complexity of the underlying problem, and the network size. Generally speaking, a larger number of data samples will do a better job at representing the underlying problem and, as long as the proper network size is used, this should allow us to learn a better solution to the problem.

The generalization issue is often viewed from two different perspectives *(Hush and Horne, 1993)*. In the first, the size of the network is fixed (presumably in accordance with the complexity of the underlying problem) and the issue becomes: ''How many training samples are required to achieve good generalization?'' This perspective is useful in applications where we have the ability to acquire as many samples as we deem necessary. In the second case, the number of training samples is fixed and the issue becomes: ''What size network gives the best generalization for this data?'' This perspective is useful in applications where we are limited in our ability to acquire training data, and we would like to know what size network is best at describing the data we have available. Both are valid view points, although the first is probably more common in the theoretical literature.

The above considerations guided much of the empirical work in this present research as described in the next chapter.

CHAPTER 6

DEVELOPMENT OF AN ANN FOR SINGLE SAMPLING PLAN DESIGN

6.1 Introduction.

As mentioned in Chapter 1, the *Larson binomial nomograph* is extensively used to design single sampling plans. Though quite efficient, it has certain shortcomings and limitations — typical of any graphical methodology. On the other hand, to design a sampling plan analytically one has to resort to a trial and error method, which is both time consuming and complex. Neural nets, due to their inherent ability to learn and then generalize the underlying relationships from the data presented to them at the time of training, were evaluated in the present work as an alternative tool to tackle this task effectively.

6.2 The Methodology Employed.

The patterns used for training the ANN were generated randomly, constrained to fall uniformly within a specified area on the nomograph (see Figure 6.1). Each pattern consists of an input vector $(1-\alpha,\ p_1,\ \beta,\ p_2)$ and an output vector $(n,\ c)$. The output vector components viz., n and c were scaled to remain within 0.1 and 0.9 in view of the fact that sigmoidal function cannot actually attain its extreme values of 0 and 1 without having infinitely large connection weights *(Rumelhart and McClelland, 1986)*.

ANNs with one hidden layer were used throughout, given the understanding that such nets are capable of forming any, possibly unbounded, convex region in the space spanned by the inputs *(Lippman, 1987)*. To choose the proper network size we adopted the strategy of *''growing a network'' (Hush and Horne, 1993)* and started with 6 and then 10 nodes in the hidden layer and later moved to 12, 15 and 18 nodes.

Study One is an attempt to achieve an optimum setting of

the controlling ANN parameters involved. It is followed by **Study Two** wherein the phenomenon of *overfitting (Leonard and Kramer, 1990)* was studied. **Study Three** went in accordance to the procedure of *''minimizing total squared error''(Leonard and Kramer, 1990)* in order to avoid overfitting and deliver an optimum ANN. The description of the experiments and results follow.

6.3 Study One: Determination of Optimum ANN Parameters

Parameters that might affect the predictive performance of an ANN are:

(1) *the learning rate (η).*

(2) *the momentum factor (α).*

(3) *the number of nodes in hidden layer (h).*

(4) *the tolerance specified (ε), and*

(5) *the number of training patterns used (P).*

In order to find the best possible settings of these parameters a parametric study was conducted as suggested in the robust design literature *(Bagchi T.P., 1993; Phadke M.S.,1989)*. Robust design methodology provides an efficient tool to evaluate factor effects under the name of Orthogonal Array-based statistical experiments. These experiments allow one to make proper decisions as regard to optimum ''factor'' settings, assuming that there exists little or no interaction between factors. This assumption may be verified later by carrying out a verification run. The orthogonal experiments and their results and implications are reported hereafter.

The "Number of Iterations" column in Table 6.1 shows the number of computing iterations needed to have the output (n and c in coded form) converge to within the specified tolerance. The pssd column displays the total pattern sum of squared deviations (pssd) value evaluated using equation,

$$\text{pssd} = \sum_{p=1}^{P} \sum_{j=1}^{q} (t_{pj} - d_{pj})^2$$

where t_{pj} is the desired activation (output value) at the output

node j for pattern p, and q is the number of output nodes *(Hwarang and Hubele, 1993)*. P refers to the number of training data used. Training patterns were used from Table 1A. The prediction pattern set P1 used in this study is given in Table 2A (see Appendix).

Orthogonal Experiment 1.

To investigate the effect of tolerance(ε) on pssd, the number of patterns used during training(P) from Table 1A and the number of nodes in the hidden layer(h), on the performance (pssd) of ANN over 10 patterns (in Table 2A), *keeping the other key parameters namely learning rate(η)* and *momentum factor(α)* fixed at *0.6 and 0.5 respectively*. See Table 6.1.

Table 6.1:

Expm.	Tolerance	No. of Patterns	Nodes	No. of Iters.	pssd on p1
1	0.05	5	6	25778	1.74782
2	0.05	5	10	22228	1.89511
3	0.05	10	6	80981	2.73743
4	0.05	10	10	66706	2.36020
5	0.10	5	6	935	0.925316
6	0.10	5	10	1257	1.16801
7	0.10	10	6	4723	1.26503
8	0.10	10	10	9589	1.23671

Table 6.2: The Response Table For Table 6.1.

Expm.	Observed	Tolerance		No. of Patterns used		No. of Nodes used	
	pssd on p1	0.05	0.10	5	10	6	10
1	1.747820	1.747820		1.747820		1.747820	
2	1.895110	1.895110		1.895110			1.895110
3	2.737430	2.737430			2.737430	2.737430	
4	2.360200	2.360200			2.360200		2.360200
5	0.925316		0.925316	0.925316		0.925316	
6	1.168010		1.168010	1.168010			1.168010
7	1.265030		1.265030		1.265030	1.265030	
8	1.236710		1.236710		1.236710		1.236710
Total	13.335626	8.740560	4.595066	5.736256	7.599370	6.675596	6.660030
# of data	8	4	4	4	4	4	4
Average	1.666953	2.185140	1.148767	1.434064	1.899843	1.668899	1.665008
Estimated main effect		1.036374		−0.465779		0.003891	

The verification run corresponds to Expm. 6 in Table 6.1 whose pssd was observed to be 1.168010. For this experiment the projected pssd assuming only main effects to be significant is,

$$= 1.666953 - \frac{1}{2}(1.036374) + \frac{1}{2}(0.465779) - \frac{1}{2}(0.003891).$$

$$= 1.666953 - 0.518187 + 0.232889 - 0.001945.$$

$$= 1.37971.$$

Thus, $\Delta = (1.37971 - 1.168010) = 0.2117$.

Therefore, it may be inferred that some interaction of factor effects do exist, but not too strong when compared to the effect of Tolerance and Number of (training) Patterns used as also shown by Figure 6.1.

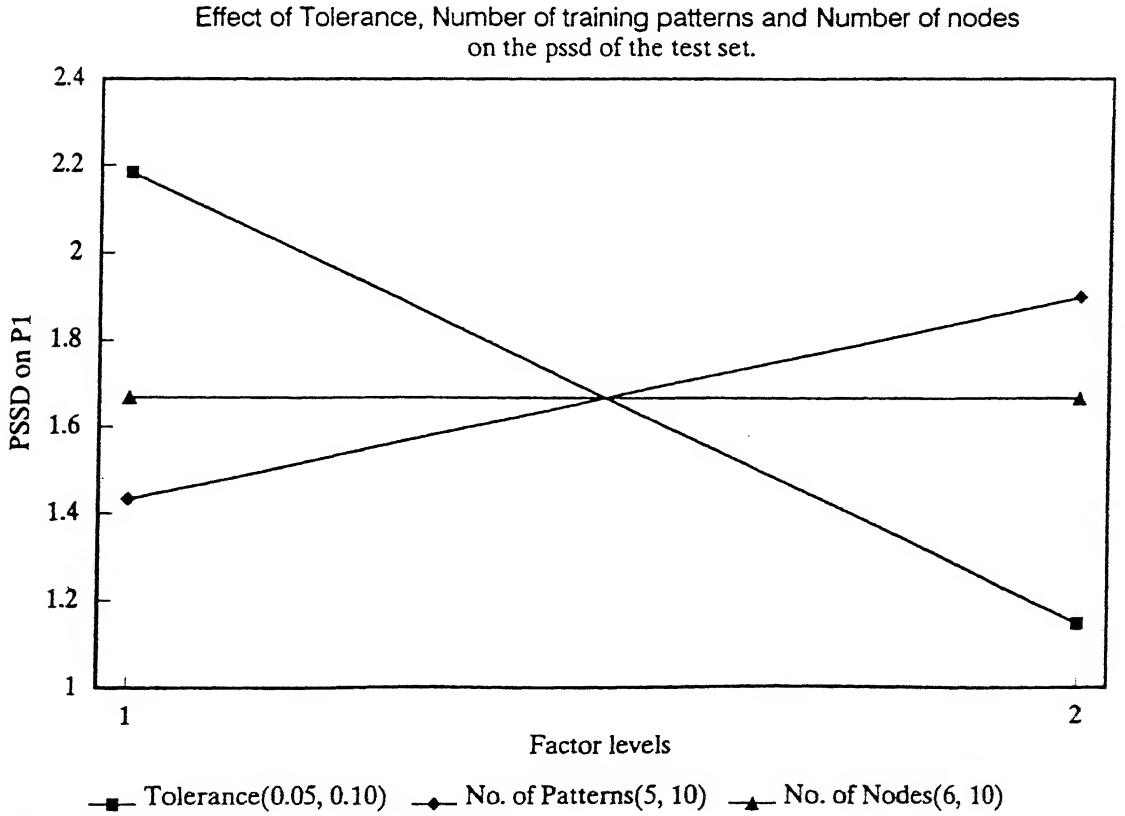


Fig.6.1: Graph showing main factor effects.

Tolerance seems to have strong effect on the performance of the net and number of patterns too seems to play significant role while the number of nodes has a weak effect.

The optimum settings of the parameters giving low pssd are therefore,

Tolerance = 0.1,

No. of Patterns = 5,

No. of Nodes = 10.

Orthogonal Experiment 2.

To investigate the effect of tolerance(ε), increase in the number of patterns used during training(P) from Table 1A and the number of nodes in the hidden layer(h) on pssd values (as defined in secn. 6.3) over 10 patterns (in Table 2A), *keeping learning rate(η) and momentum factor(α)* fixed at *0.6 and 0.5 respectively*, factor settings as shown in Table 6.3 were used.

Table 6.3:

Expm.	Tolerance.	# Patterns	Nodes	No. of Iters.	pssd on p1
1	0.05	20	6	127442	1.31735
2	0.05	20	10	58473	1.64862
3	0.05	40	6	Too high	1.76148
4	0.05	40	10	Too high	1.71660
5	0.10	20	6	30389	1.66261
6	0.10	20	10	37766	1.65557
7	0.10	40	6	Too high	1.63949
8	0.10	40	10	Too high	1.54435

Table 6.4: The Response Table For Table 6.3.

Expm.	Observed	Tolerance		No. of Patterns		No. of Nodes	
	pssd on p1	0.05	0.10	20	40	6	10
1	1.317350	1.317350		1.317350		1.317350	
2	1.648620	1.648620		1.648620			1.648620
3	1.761480	1.761480			1.761480	1.761480	
4	1.716600	1.716600			1.716600		1.716600
5	1.662610		1.662610	1.662610		1.662610	
6	1.655570		1.655570	1.655570			1.655570
7	1.639490		1.639490		1.639490	1.639490	
8	1.544350		1.544350		1.544350		1.544350
Total	12.946070	6.444050	6.502020	6.284150	6.661920	6.380930	6.565140
# of data	8	4	4	4	4	4	4
Average	1.618259	1.611013	1.625505	1.571038	1.665480	1.595233	1.641285
Estimated main effect		−0.014492		−0.094443		−0.046053	

Experiment 1 in Table 6.3 incidentally corresponds to the verification run we ought to do. Its observed pssd was 1.31735. For this experiment the projected pssd assuming only main effects to be significant is,

$$= 1.618259 + \frac{1}{2}(0.014492) + \frac{1}{2}(0.094443) + \frac{1}{2}(0.046053).$$

$$= 1.618259 + 0.007494 + 0.047222 + 0.023027.$$

$$= 1.696002.$$

Thus, $\Delta = (1.696002 - 1.31735) = 0.378652.$

Therefore, the presence of interactions cannot be ruled out. The influence of these parameters on the pssd is depicted in the following graph.

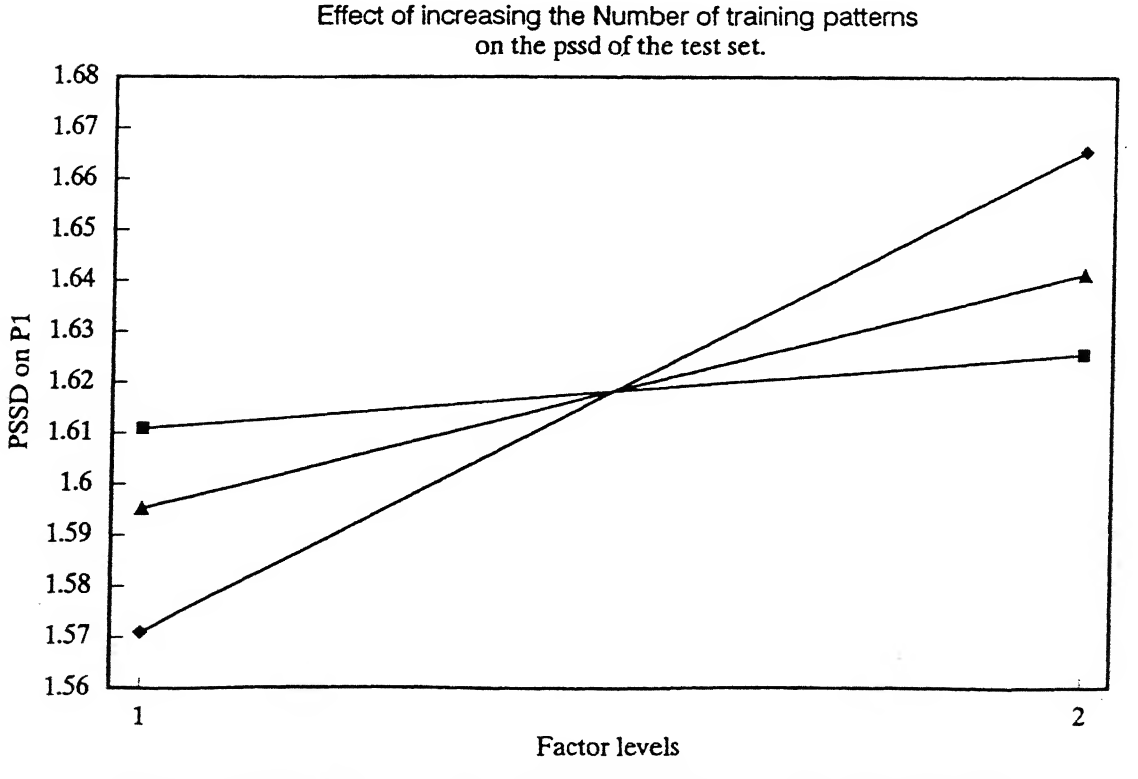


Fig. 6.2: Graph showing main factor effects.

Tolerance seems to have lost its influence, the effect of the number of (training) patterns has strengthened, and the number of nodes has become significant as evidenced by Figure 6.2.

The optimum settings for the parameters to minimize pssd are therefore projected to be,

Tolerance = 0.05,

No. of Patterns = 20,

No. of Nodes = 6.

Orthogonal Experiment 3.

To investigate the effect of tolerance(ε), learning rate(η) and momentum factor(α) on the performance (pssd, ref. secn. 6.3) of *an ANN having 6 nodes in the hidden layer trained upon 10 patterns (of Table 1A)* factor settings as shown in Table 6.5 were used.

Table 6.5:

Expm.	Tolerance.	Learning rt.	Momentum	No. of Iters	pssd on p1
1	0.05	0.6	0.5	80981	2.73743
2	0.05	0.9	0.5	33395	2.38751
3	0.05	0.6	0.8	19941	2.29234
4	0.05	0.9	0.8	19170	2.98982
5	0.10	0.6	0.5	4723	1.26503
6	0.10	0.9	0.5	3362	1.30055
7	0.10	0.6	0.8	2250	1.33534
8	0.10	0.9	0.8	2114	1.39410

Table 6.6: The Response Table For Table 6.5.

Expm.	Observed	Tolerance		Momentum factor		Learning rate	
	pssd on p1	0.05	0.10	0.5	0.8	0.6	0.9
1	2.737430	2.737430		2.737430		2.737430	
2	2.387510	2.387510		2.387510			2.387510
3	2.292340	2.292340			2.292340	2.292340	
4	2.989820	2.989820			2.989820		2.989820
5	1.265030		1.265030	1.265030		1.265030	
6	1.300550		1.300550	1.300550			1.300550
7	1.335340		1.335340		1.335340	1.335340	
8	1.394100		1.394100		1.394100		1.394100
Total	15.702120	10.407100	5.295020	7.690520	8.011600	7.630140	8.071980
# of data	8	4	4	4	4	4	4
Average	1.962765	2.601775	1.323755	1.922630	2.002900	1.907535	2.017995
Estimated main effect		1.278020		−0.080270		−0.110460	

The verification run corresponds to Expm. 5 in the above tableau whose observed pssd is 1.26503. For this experiment the projected pssd assuming only main effects to be significant is,

$$= 1.962765 - \frac{1}{2}(1.278020) + \frac{1}{2}(0.080270) + \frac{1}{2}(0.110460).$$

$$= 1.962765 - 0.63901 + 0.040135 + 0.05523.$$

$$= 1.41912.$$

Therefore, $\Delta = (1.41912 - 1.26503) = 0.15409$.

Thus, here the interactions seem to have relatively little influence as compared to the experiments performed earlier. Therefore, we infer that the value of tolerance should not be too tight (possibly because that leads to the phenomenon of *overtraining* explored in Study Two), and the learning rate and momentum factor must be assigned the smallest value feasible (*keeping in view the practical implications of retarded rate of training*). The effects are graphically shown in Figure 6.3.

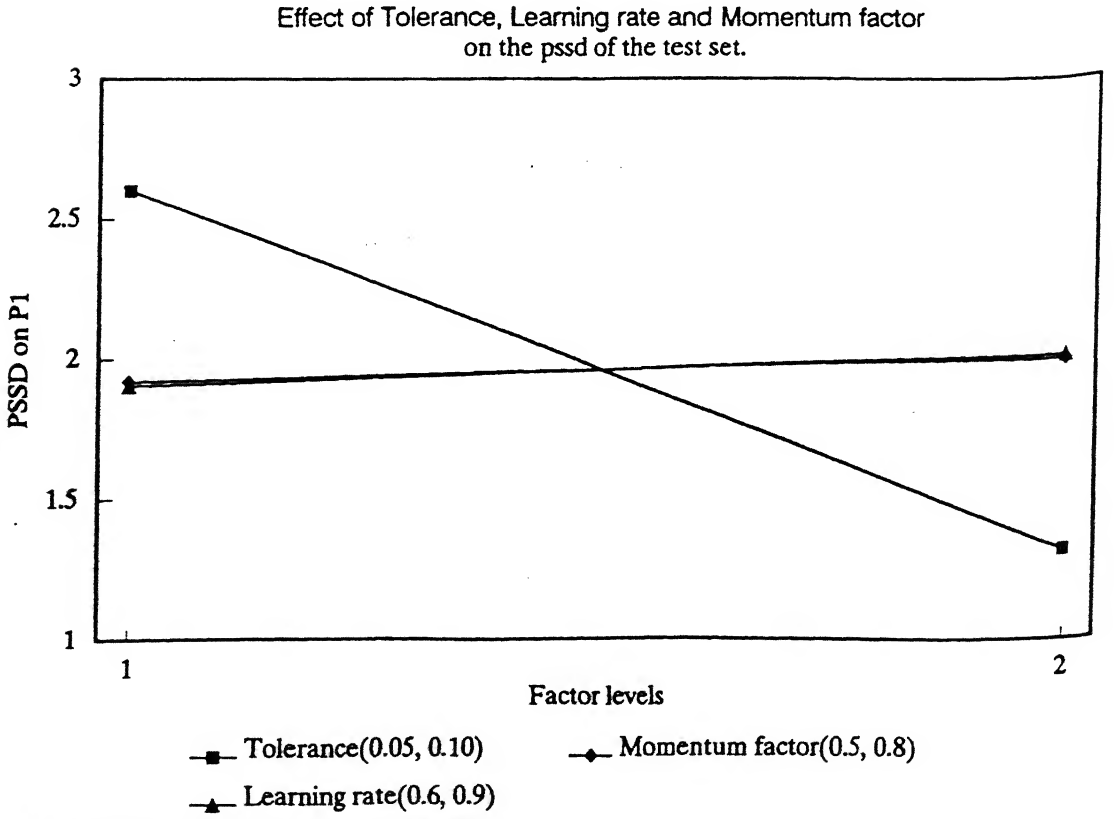


Fig. 6.3: Graph showing main factor effects.

The optimum parameter settings to lead to smallest pssd therefore are,

Tolerance = 0.1,

Learning rate = 0.6,

Momentum factor = 0.5.

6.4 Study Two: Monitoring the ''Overtraining''

The above sets of experiments indicated the directional effects of the parameters only. We next turned to apply the technique of *cross validation* to optimize the ANN *(Hush and Horne, 1993)*. The method essentially consists of dividing the sample set into two subsets, one used in *training* the network and the other to keep *a close watch* on the generalizing ability of the net. The training of the net is terminated at a point when the net becomes sufficiently capable of *generalizing* (please see secn. 5.4).

Seperate experiments were carried using ANNs having 6, 12 and 15 nodes (in the intermediate layer) respectively. The results obtained are reported in subsequent sections (in the order they were carried out). The training patterns used were from Table 1A. P1 and P2 refer to the prediction set given in Table 2A and Table 3A (Appendix) respectively. The total pssd is evaluated as earlier using equation,

$$\text{pssd} = \sum_{p=1}^{P} \sum_{j=1}^{q} (t_{pj} - d_{pj})^2$$

where, t_{pj} is the desired activation (output value) at the output node j for pattern p, and q is the number of output nodes *(Hwarang and Hubele, 1993)*. P refers to the number of training data used.

Experiment I.

An experiment using a 15-node ANN trained on 50 patterns (Table 1A), *keeping the learning rate(η) and the momentum factor fixed at o.6 and 0.5 respectively* was conducted See Table 6.7. Pssd of two sets of 10 patterns each (in Table 2A and 3A) were monitored.

Table 6.7:

Iters done	pssd on P1	pssd on P2
1	0.959149	1.09697
10000	0.729204	0.044246
20000	0.532398	0.15309
30000	0.58794	0.181064
40000	0.728215	0.277197
50000	0.834858	0.342957
60000	0.702366	0.380695
70000	0.665619	0.503793
80000	0.586661	0.505648
90000	0.529464	0.534738

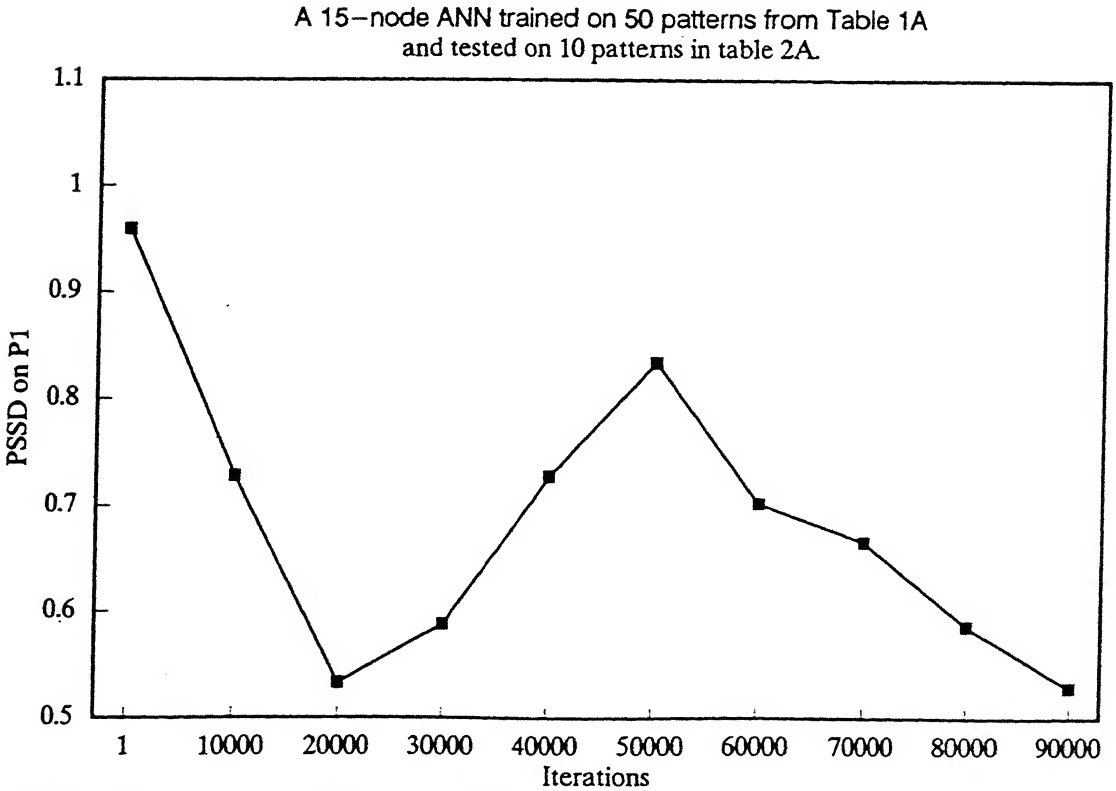


Fig. 6.4: Pssd vs Iters. Learning rt. = 0.6, Momentum factor = 0.5.
Minima of 0.529464 occurs after 90000 iterations.

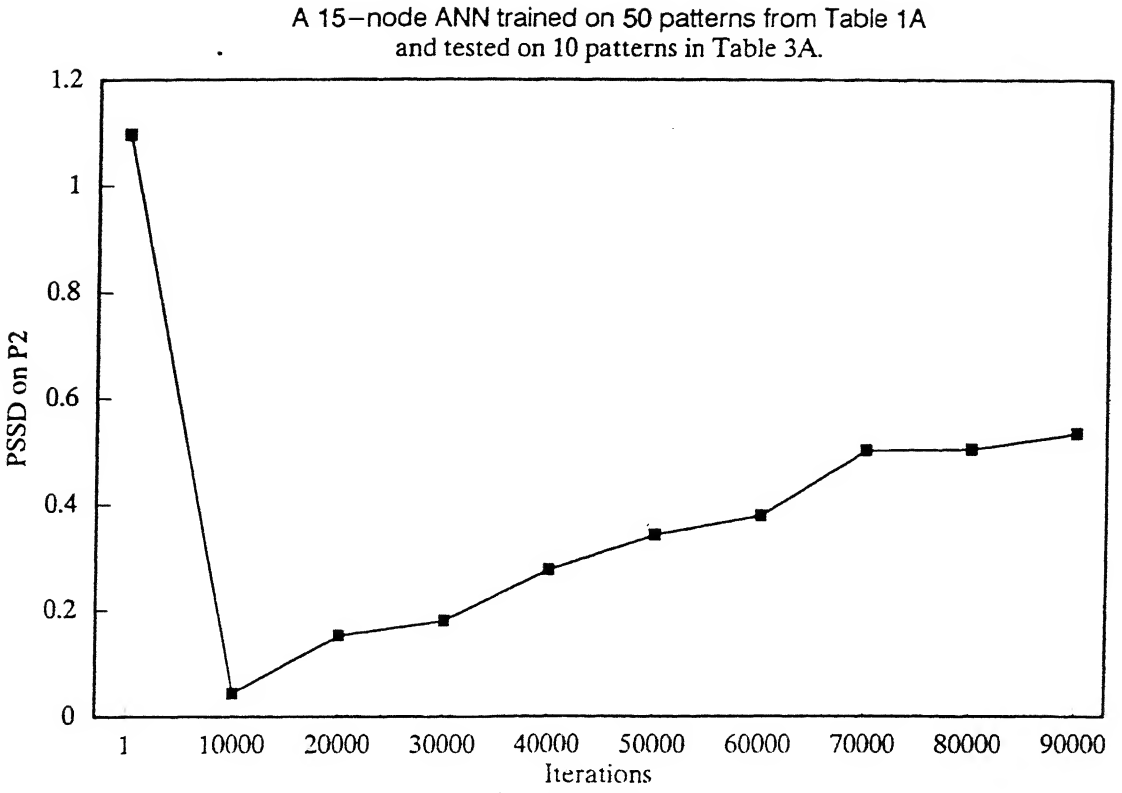


Fig. 6.5: Pssd vs Iters. Learning rt. = 0.6, Momentum factor = 0.5.
Minima of 0.044246 occurs after 10000 iterations.

Experiment II.

Experiment was conducted on a 12-node ANN trained on 50 patterns, *keeping the learning rate(η) and the momentum factor(α) fixed at 0.6 and 0.5 respectively*. See Table 6.8. Pssd was monitored as done earlier using 10 patterns shown in Tables 2A and 3A.

Table 6.8:

Iters done	pssd on P1	pssd on P2
1000	0.519449	0.362156
10000	0.441796	0.036145
20000	0.613458	0.240482
30000	0.657618	0.237533
40000	0.786305	0.270671
50000	0.946507	0.346091
60000	1.01452	0.492297
70000	1.11009	0.595601
80000	1.13193	0.524476
90000	1.13521	0.518867
94500	1.137791	0.522885

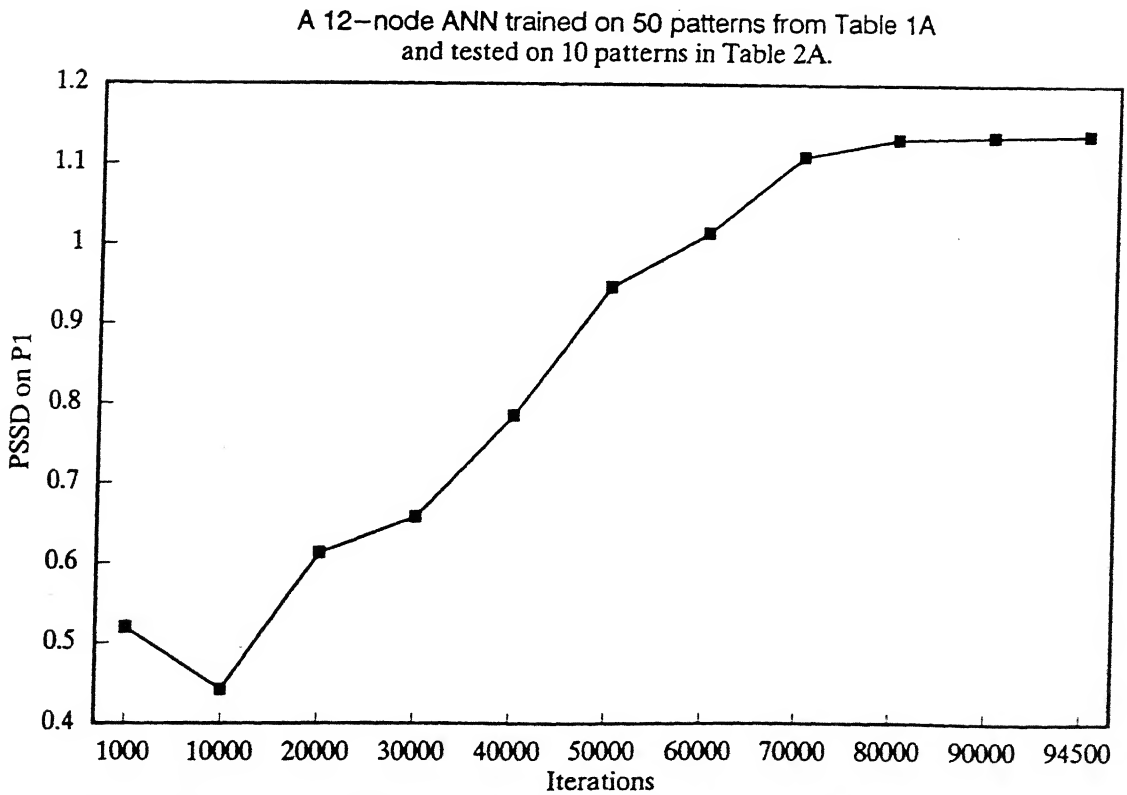


Fig.6.6:Pssd vs Iters. Lrng rt. 0.6,Momtm. 0.5. This configuration is probably grossly overtrained as it fails badly in its ability to generalize. Minima of 0.441796 occurs after 10000 iterations.

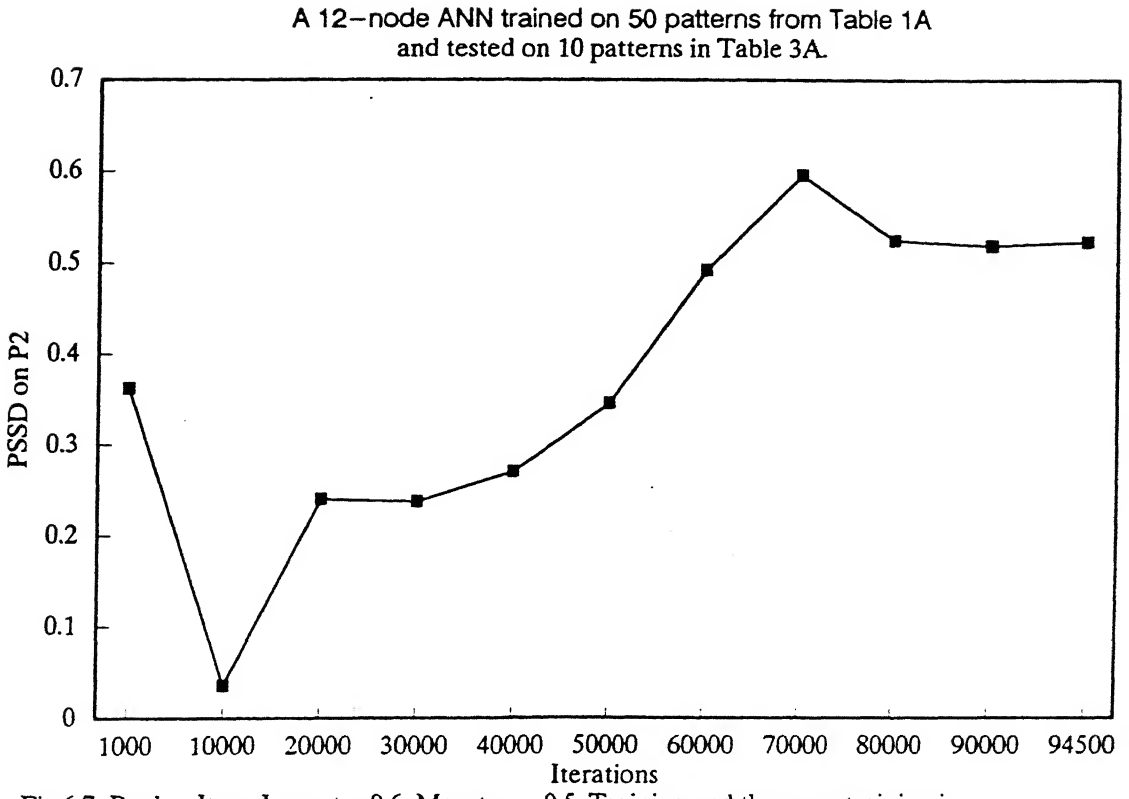


Fig.6.7: Pssd vs Iters. Lrng rt.=0.6, Momtm.= 0.5. Training and then overtraining is apparent. Minima of 0.036145 occurs after 10000 iterations.

Experiment III.

Experiment on a 6-nodes ANN trained over 10 patterns (Table 1A), *learning rate and momentum factor being fixed at 0.6 and 0.5 respectively*. See Table 6.9. Pssd was over 10 patterns (in Table 2A and 3A) monitored.

Table 6.9:

Iters done	pssd on P1	pssd on P2
1	0.84024	1.13394
100	0.61236	0.498976
200	0.620518	0.523371
300	0.66111	0.569604
400	0.688582	0.592062
499	0.702613	0.598971
500	0.703012	0.599251
1000	0.762669	0.762669
1500	0.837803	0.601053
2000	0.940765	0.579502
2500	1.06852	0.515134
3000	1.15346	0.453074
3500	1.18985	0.417111
4000	1.121984	0.40474
4500	1.25095	0.403644
5000	1.26565	0.405209
5500	1.26556	0.40563

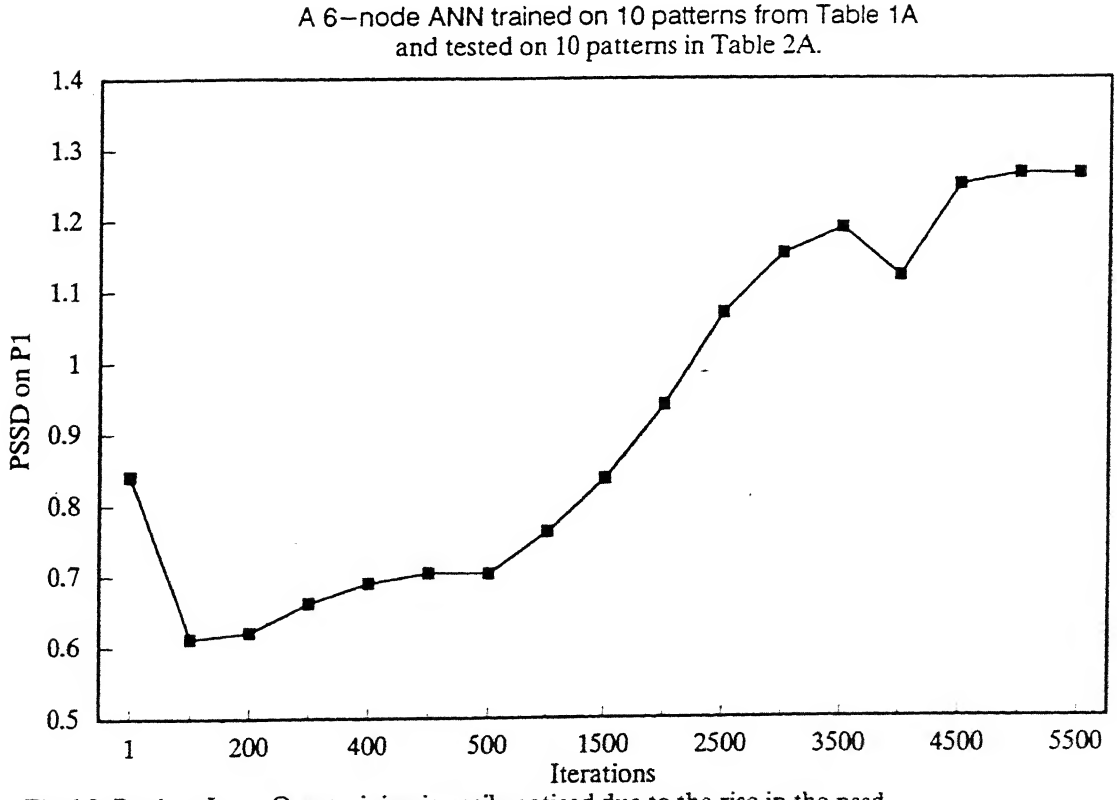


Fig.6.8: Pssd vs. Iters. Overtraining is easily noticed due to the rise in the pssd curve as the training proceeds. Minima of 0.61236 occurs at 100th iteration.

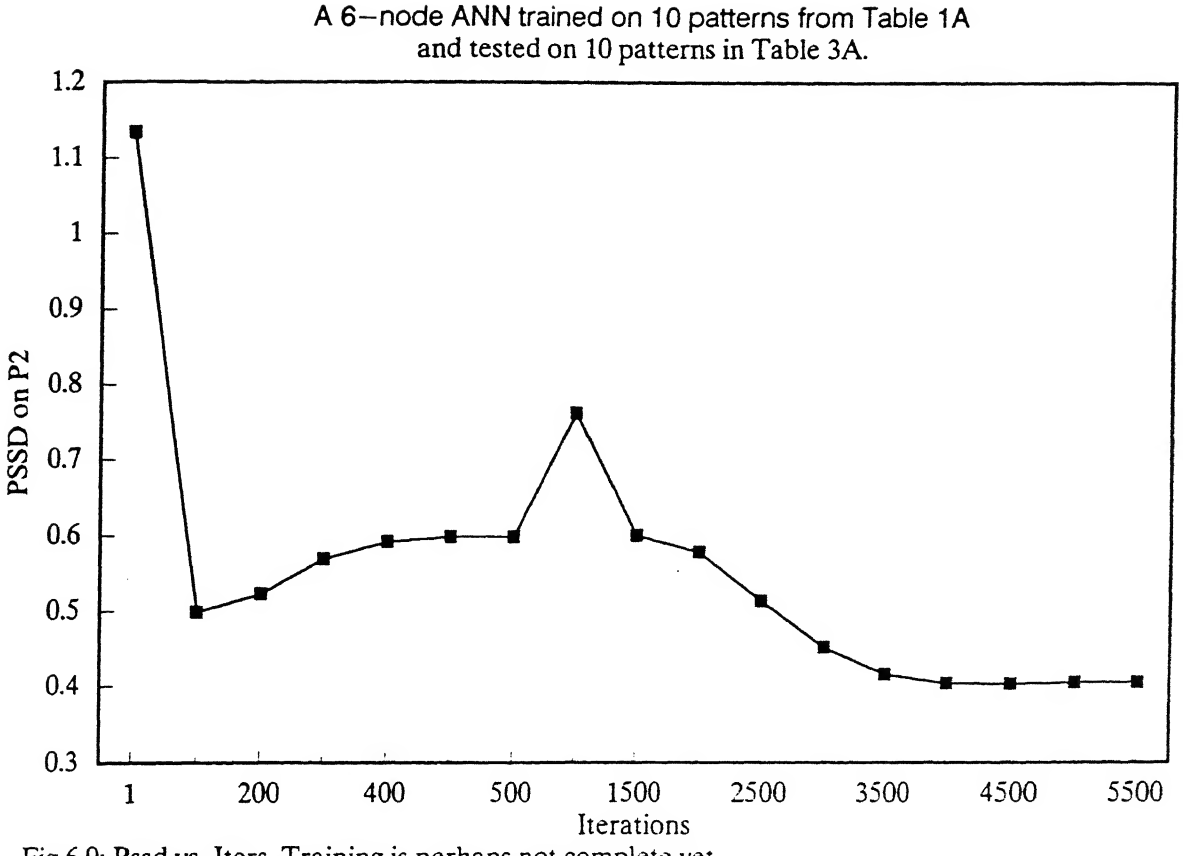


Fig.6.9: Pssd vs. Iters. Training is perhaps not complete yet.
Minima of 0.403644 occurs at 4500 iteration.

Experiment IV.

Experimentation with 6-nodes ANN trained on 20 patterns (Table 1A), *learning rate(η) and momentum factor(α) being at 0.6 and 0.5 respectively*. See Table 6.10. Pssd was over 10 patterns (in Table 2A and 3A) monitored.

Table 6.10:

Iters done	pssd on P1	pssd on P2
1	1.07783	1.58745
500	0.659187	0.649129
1000	0.701829	0.576425
1500	0.728589	0.527988
2000	0.775077	0.541257
2500	0.841029	0.571625
3000	0.990651	0.649981
3500	1.16546	0.753341
4000	1.24673	0.789552
4500	1.29918	0.799555
5000	1.34582	0.81311
5500	1.43202	0.8298
6000	1.50955	0.829304
6500	1.5608	0.805155
7000	1.5898	0.768436
7500	1.60032	0.728263
8000	1.5869	0.665821
8500	1.55371	0.582924
9000	1.51685	0.497413
9500	1.50079	0.433762
10000	1.55039	0.438686
10500	1.65255	0.498544
11000	1.72594	0.513555
11500	1.79764	0.499492
12000	1.88199	0.491613
12500	1.98047	0.509093
13000	2.06596	0.528345
13500	2.11067	0.519056
14000	2.11743	0.48687
14500	2.10737	0.45387
15000	2.09493	0.432157
15500	2.08437	0.42016
16000	2.07677	0.414794
16500	2.07029	0.411617
17000	2.0611	0.40854
17500	2.05168	0.405946
18000	2.04121	0.403156
18500	2.02649	0.400671
19000	2.0123	0.398657
19500	1.99698	0.396283
20000	1.98222	0.394289
20500	1.96799	0.391537
21000	1.9531	0.385631
21500	1.95598	0.30054
22000	1.89049	0.393123
22500	1.87332	0.40203
23000	1.86104	0.391184
23500	1.87069	0.337473
24000	1.77114	0.343722
24500	1.80372	0.389878
25000	1.87502	0.326812
25500	1.77385	0.36131
26000	1.69777	0.370326
26500	1.81687	0.316713
27000	1.71814	0.34243
27500	1.64135	0.339597
28000	1.70602	0.325681
28500	1.64715	0.33465
29000	1.66803	0.334996
29500	1.66412	0.334137
30000	1.66192	0.337822
30500	1.6618	0.34096
31000	1.66115	0.3436
31500	1.66161	0.344756

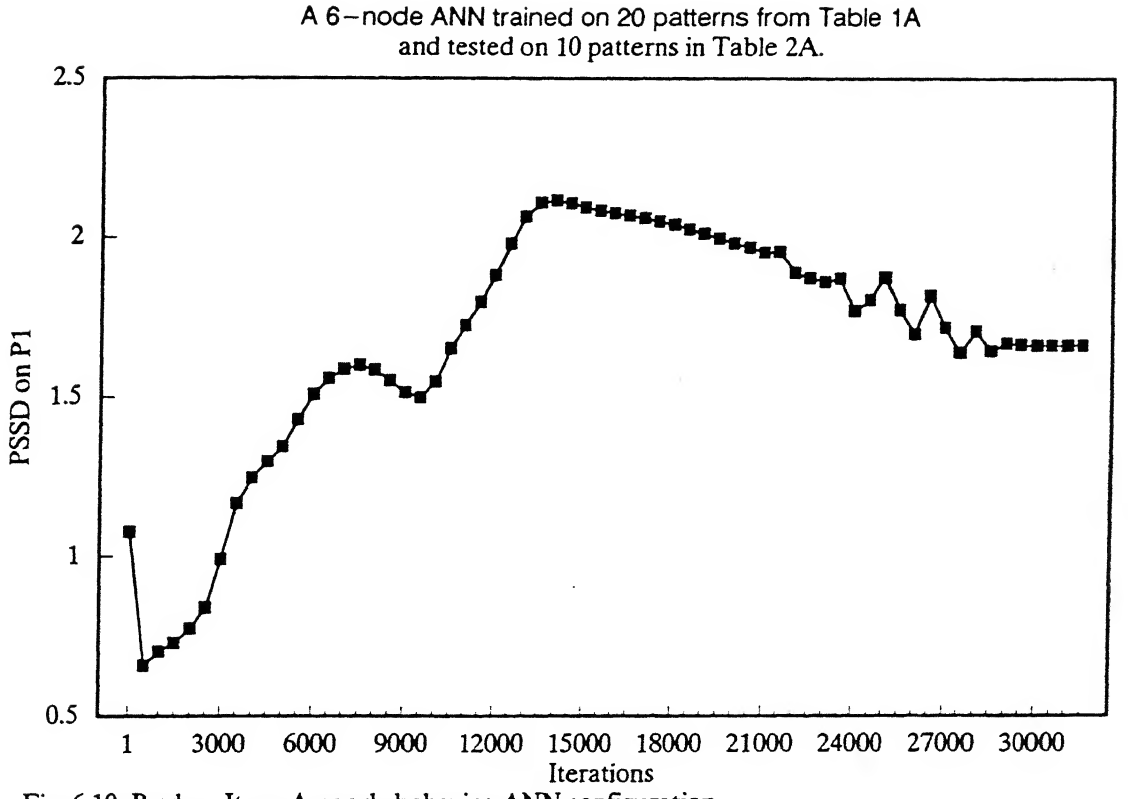


Fig. 6.10: Pssd vs. Iters. A poorly behaving ANN configuration.
Lrng. rt 0.6,Momtm. 0.5.Minima of 0.659187 occurs at 500 iteration.

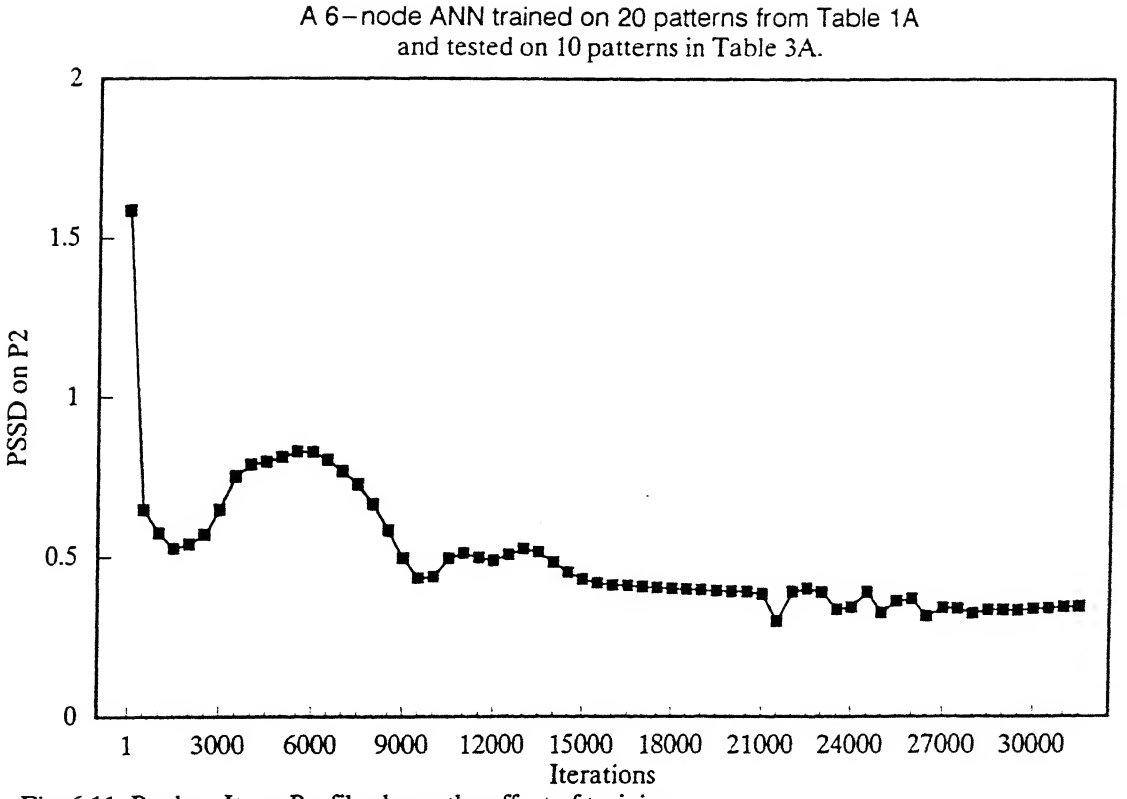


Fig. 6.11: Pssd vs. Iters. Profile shows the effect of training.
Lrng rt. 0.6,Momtm. 0.5.Minima of 0.30054 occurs at 21500 iteration.

The overall tendency of the ANN to train upto a certain number of iterations (when pssd decreases to a minimum) and then ''overtrain'' (the failure of the ANN to generalize over unseen patterns) is clearly noticeable in each of the above experiments.

6.5 STUDY THREE: Minimization of Total Squared Error.

In the experiments described in the previous section the tendency of the ANN to ''overtrain'' became rapidly apparent. Further, training on a training set and then monitoring the pssd on a separate prediction set might work well with problems involving classification of samples *(Leonard and Kramer, 1990)*, it does not appears suitable for applications (such as the design of acceptance sampling plans) that require prediction (generalization from the results of training).

Leonard and Kramer *(1990)* suggest an alternative strategy to train such nets. Their approach consists of partitioning the training set into two equally (preferably) sized subsets **M'** and **M''**. **M'** is the set used to train the network and **M''** is one which is held back to test the generalization ability of the resulting net. An ensemble of networks with different set partitionings and different initial conditions $\mathbf{w}_o$ are trained using E(**M'**), the error over the training set,as the objective for improving. Training runs are halted at the point where E(**M**) = E(**M'**) + E(**M''**), the error over the combined set,gets minimized. Typically, the network configuration that produces the lowest value of E(**M**) is selected. The study in this section was carried out based on this strategy.

Assuming that we have a finite number of samples at our disposal(say 100), we would train our net by partitioning the sample set into two subsets of 50 samples each. For our application the sample points were randomly generated on the nomograph (shown in Table 4A and Figure 2A). Table 5A gives the test set on which the generalization ability of the net was assessed which is also shown in Figure 2A along with the training points. ANNs having 6, 12 and 18 nodes were studied. The experiments and their results are summarized in this section.

Experiment V.

Experiment on a 6-node ANN trained on 50 patterns of the training set (Table 4A) and its performance judged on 50 patterns

(in Table 5A) of the test set. *The learning rate(η) and the momentum factor(α) were fixed at 0.6 and 0.5 respectively.* (Table 6.11).

Table 6.11:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	1.00186	3.52099	4.52285
20000	0.817872	4.39317	5.211042
30000	0.772933	4.21272	4.985653
40000	0.727196	4.2279	4.955096
50000	0.705284	4.42663	5.131914
60000	0.697067	4.58346	5.280527
70000	0.694248	4.7126	5.406848
80000	0.693437	4.80728	5.500717
90000	0.693388	4.86366	5.557048
100000	0.693595	4.89701	5.590605

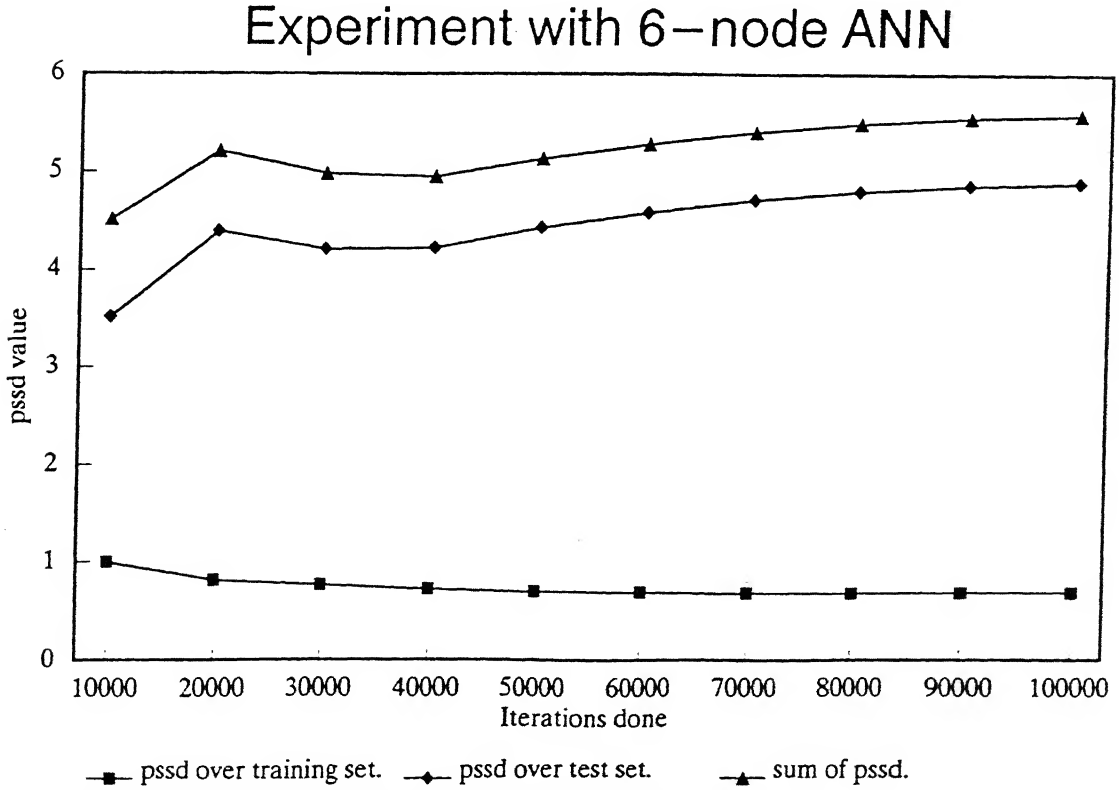


Fig. 6.12: Variation of cumulative pssd with iters for 6 nodes ANN.
The training set and test set each had 50 patterns. Minima of 4.52285 at 10000 iters.

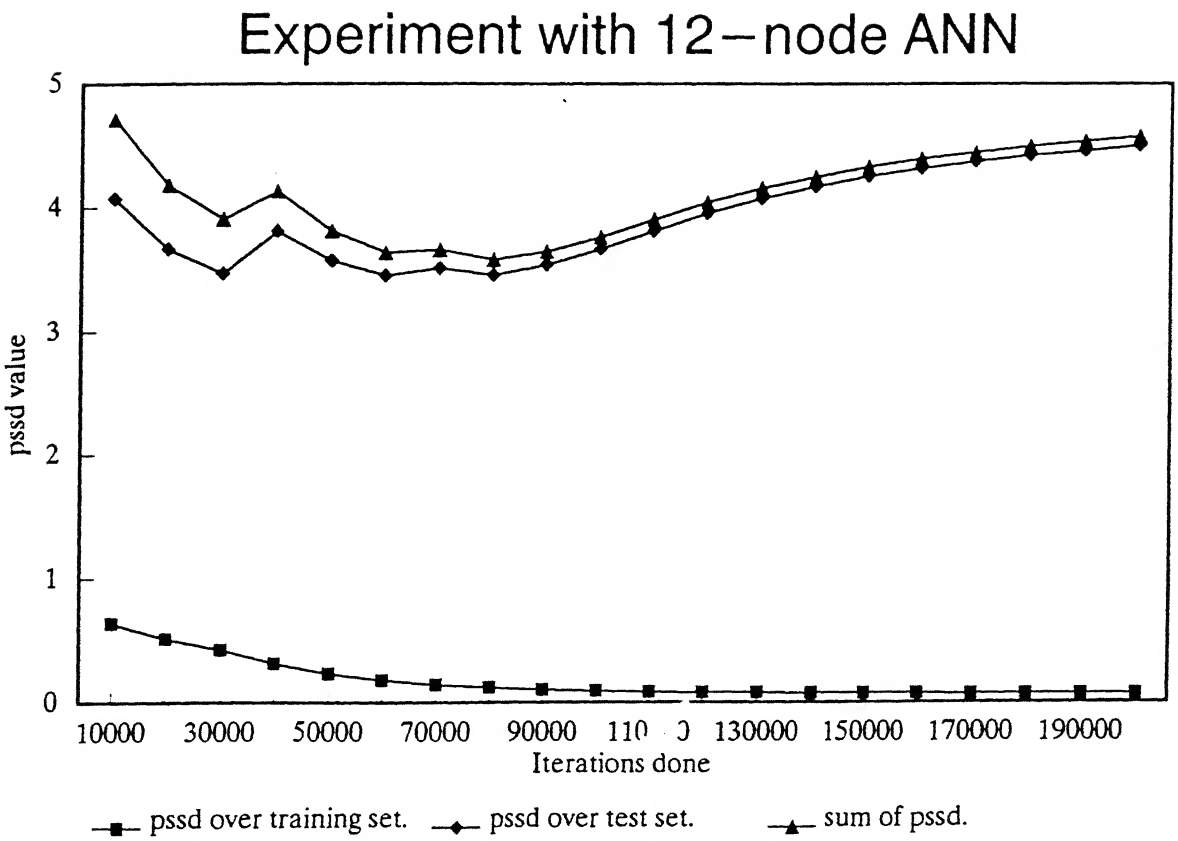


Fig. 6.13: Variation of cumulative pssd with iters for a 12 nodes ANN.
The training set and test set each had 50 patterns. Minima of 3.58706 at 80000 iters.

Experiment VII.

Experiment on an ANN having 18 nodes trained on 50 patterns (of Table 4A) and its performance judged on 50 patterns (in Table 5A) of the test set. *The learning rate(η) and momentum factor(α) were fixed at 0.6 and 0.5 respectively.* (Table 6.13).

Table 6.13:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	0.58836	4.10427	4.69263
20000	0.413076	4.84281	5.255886
30000	0.241163	5.3154	5.556563
40000	0.122566	5.33442	5.456986
50000	0.074989	5.4745	5.549489
60000	0.0536852	5.86843	5.9221152
70000	0.0377578	6.34072	6.3784778
80000	0.0276816	6.74258	6.7702616
90000	0.021955	6.93228	6.954235
100000	0.0208942	7.14235	7.1632442

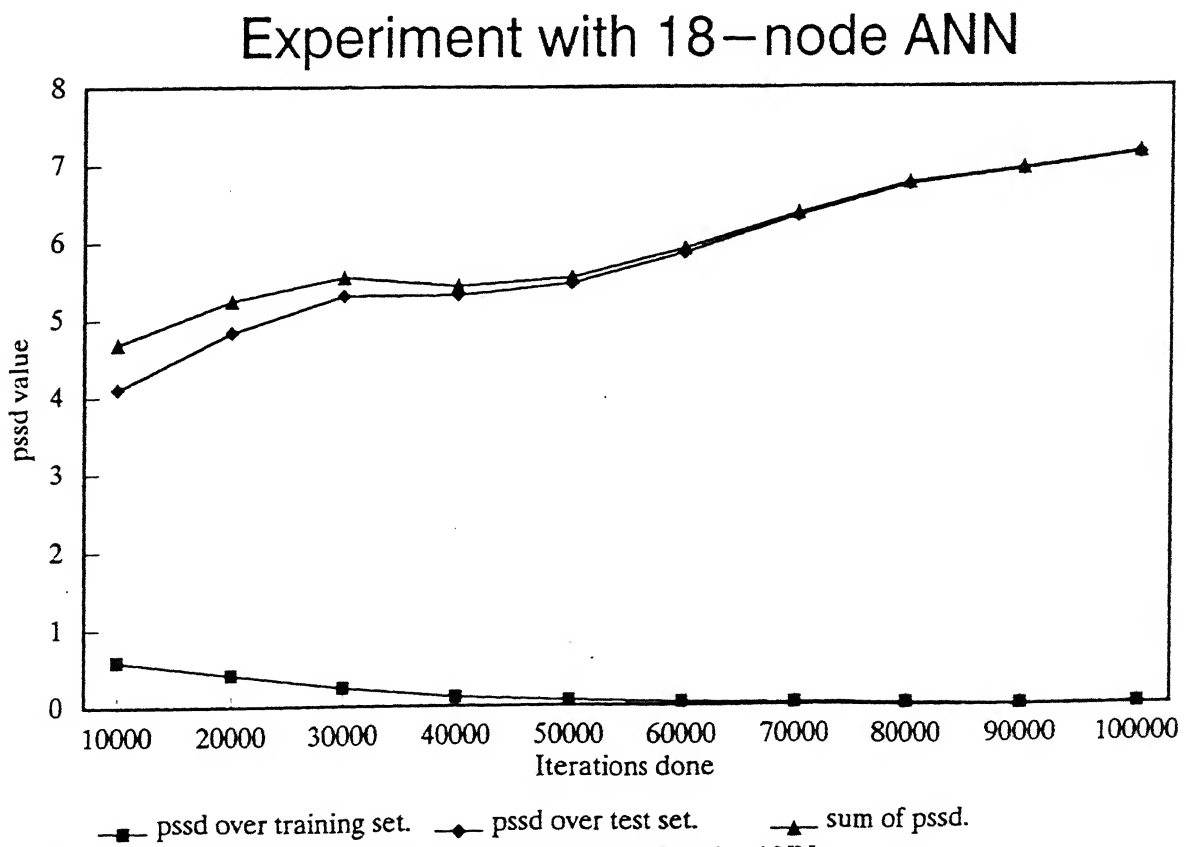


Fig. 6.14: Variation of cumulative pssd with iters for 18 nodes ANN.
The training set and test set each had 50 patterns. Minima of 4.69263 at 10000 iters.

6.5.2 Partitioning of the set into two equally halved sets was carried out in the earlier section assuming that the sample patterns available to us was finite (100 patterns). In reality, we have an unlimited number of sample points at our disposal. Therefore, here we adopt the strategy of finding out the number of training patterns that would be required to provide the best (*minimum*) pssd over the test set, given a net with a fixed number of nodes in its hidden layer. ANNs having 12, 18 and 6 nodes were tested respectively. Learning rate of 0.6 and momentum factor of 0.5 was used during the study.

Experiment VIII.

Experiment with 12 nodes ANN trained on 50 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. (Table 6.14).

Table 6.14:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	0.642402	1.46039	2.102792
20000	0.517909	1.4292	1.947109
30000	0.433647	1.33605	1.769697
40000	0.319465	1.50525	1.824715
50000	0.240205	1.52469	1.764895
60000	0.182884	1.39064	1.573524
70000	0.148415	1.38373	1.532145
80000	0.12251	1.32329	1.4458
90000	0.106152	1.29773	1.403882
100000	0.094528	1.26613	1.360658
110000	0.0877204	1.26036	1.3480804
120000	0.0836397	1.27474	1.3583797
130000	0.0810555	1.2902	1.3712555
140000	0.0792285	1.30175	1.3809785
150000	0.0778937	1.31136	1.3892537
160000	0.0768857	1.31761	1.3944957
170000	0.0759741	1.32243	1.3984041
180000	0.0751871	1.32806	1.4032471
190000	0.0744922	1.33164	1.4061322
200000	0.0738272	1.33558	1.4094072

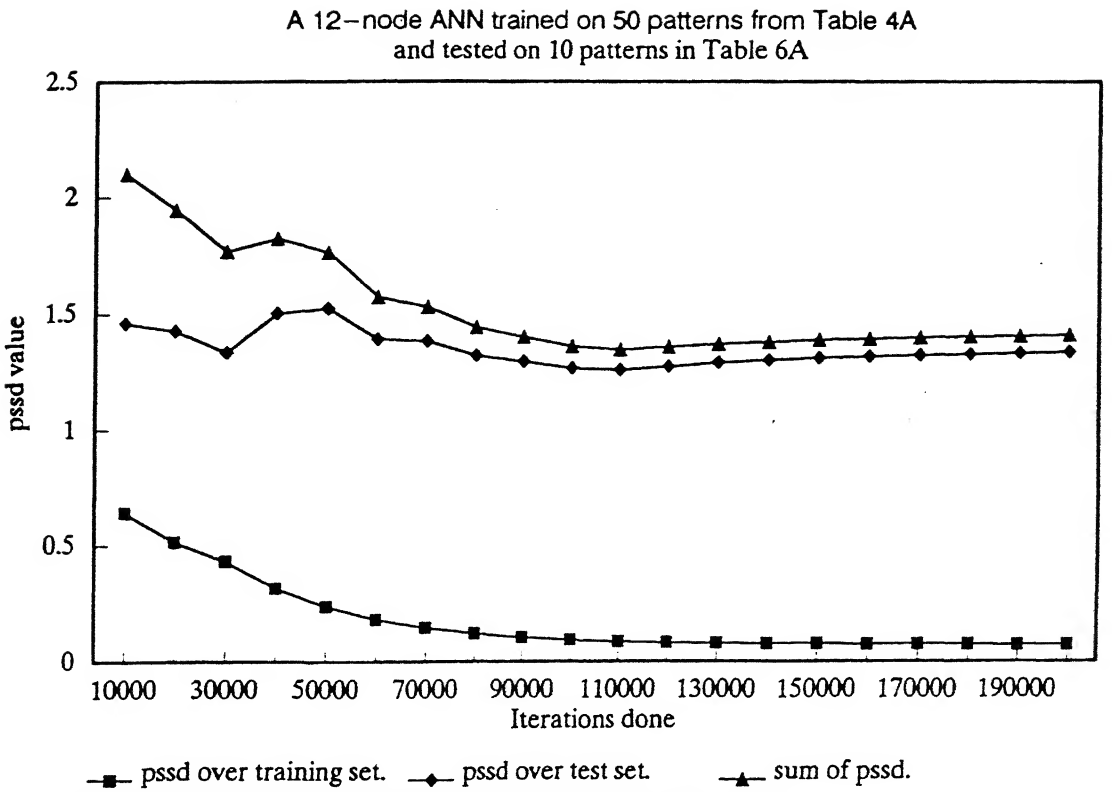


Fig. 6.15: Variation of cumulative pssd with iters for a 12 nodes ANN.
The training set/test set had 50/10 patterns. Minima of 1.3480804 at 110000 iters.

Experiment IX.

Experiment with 12 nodes ANN trained on 75 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. (Table 6.15).

Table 6.15:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	0.820805	0.559677	1.380482
20000	0.624713	0.569949	1.194662
30000	0.534278	0.419846	0.954124
40000	0.435489	0.371763	0.807252
50000	0.396789	0.370918	0.767707
60000	0.378566	0.375166	0.753732
70000	0.368025	0.380113	0.748138
80000	0.361139	0.385773	0.746912
90000	0.356437	0.389405	0.745842
100000	0.353188	0.392314	0.745502
110000	0.350769	0.39427	0.745039
120000	0.348881	0.396554	0.745435
130000	0.346847	0.399137	0.745984
140000	0.343835	0.401847	0.745682
150000	0.340502	0.407033	0.747535

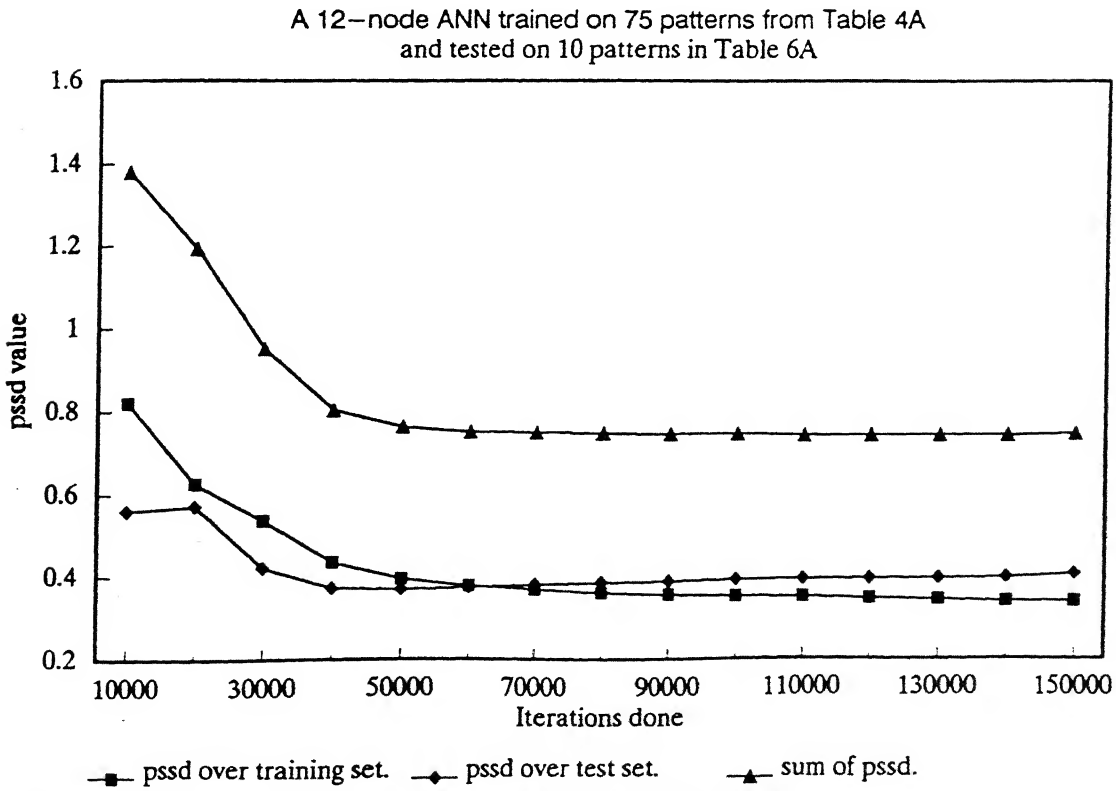


Fig. 6.16: Variation of cumulative pssd with iters for a 12 nodes ANN.
The training set/test set had 75/10 patterns. Minima of 0.745039 at 110000 iters.

Experiment X.

Experiment with 12 nodes ANN trained on 100 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. (Table 6.16).

Table 6.16:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	1.88966	0.262071	2.151731
20000	1.37262	0.23754	1.61016
30000	1.05678	0.264448	1.321228
40000	0.89298	0.284539	1.177519
50000	0.799402	0.323219	1.122621
60000	0.739834	0.345652	1.085486
70000	0.709696	0.376911	1.086607
80000	0.677611	0.4307	1.108311
90000	0.645942	0.485336	1.131278
100000	0.637002	0.536053	1.173055
110000	0.631559	0.580553	1.212112
120000	0.625342	0.613746	1.239088
130000	0.618588	0.637151	1.255739

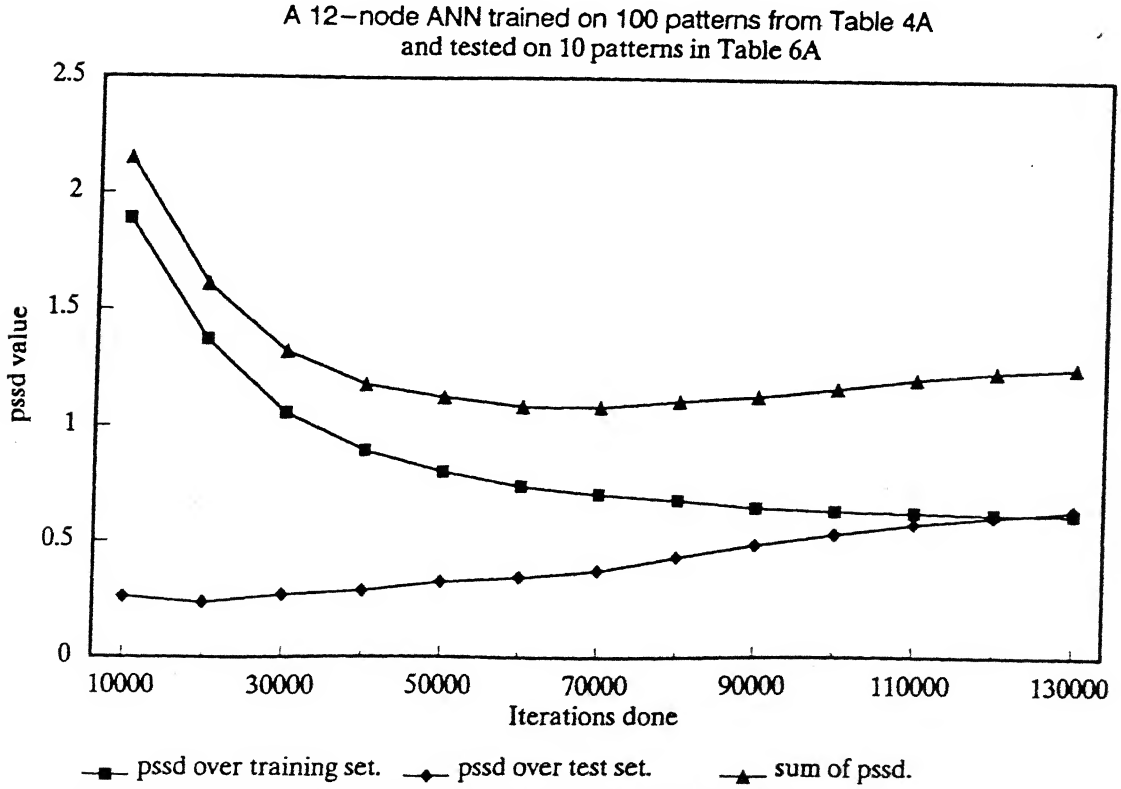


Fig. 6.17: Variation of cumulative pssd with iters for a 12 nodes ANN.
The training set/test set had 100/10 patterns. Minima of 1.085486 at 60000 iters.

Experiment XI.

Experiment with 18 nodes ANN trained on 50 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. (Table 6.17).

Table 6.17:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	0.58836	2.02779	2.61615
20000	0.413076	2.769196	3.182272
30000	0.241163	2.485923	2.727086
40000	0.122566	2.251646	2.374212
50000	0.074989	2.273769	2.348758
60000	0.0536852	2.30822	2.3619052

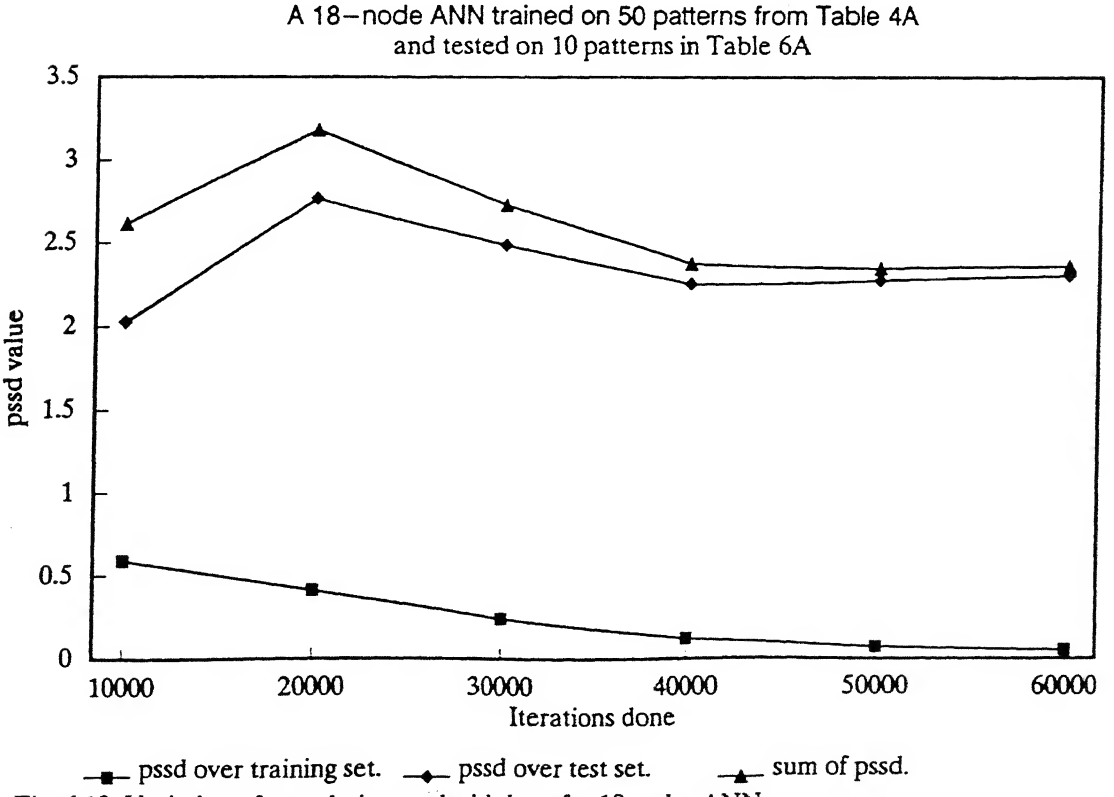


Fig. 6.18: Variation of cumulative pssd with iters for 18 nodes ANN.
The training set/test set had 50/10 patterns. Minima of 2.02779 at 10000 iters.

Experiment XII.

Experiment with 18 nodes ANN trained on 75 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. (Table 6.18).

Table 6.18:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	0.92073	0.17704	1.09777
20000	0.709818	0.29095	1.000768
30000	0.557139	0.219133	0.776272
40000	0.455083	0.246331	0.701414
50000	0.386286	0.332669	0.718955
60000	0.309636	0.393938	0.703574
70000	0.235808	0.438024	0.673832
80000	0.203807	0.513837	0.717644
90000	0.159479	0.663347	0.822826
100000	0.127909	0.68733	0.815239

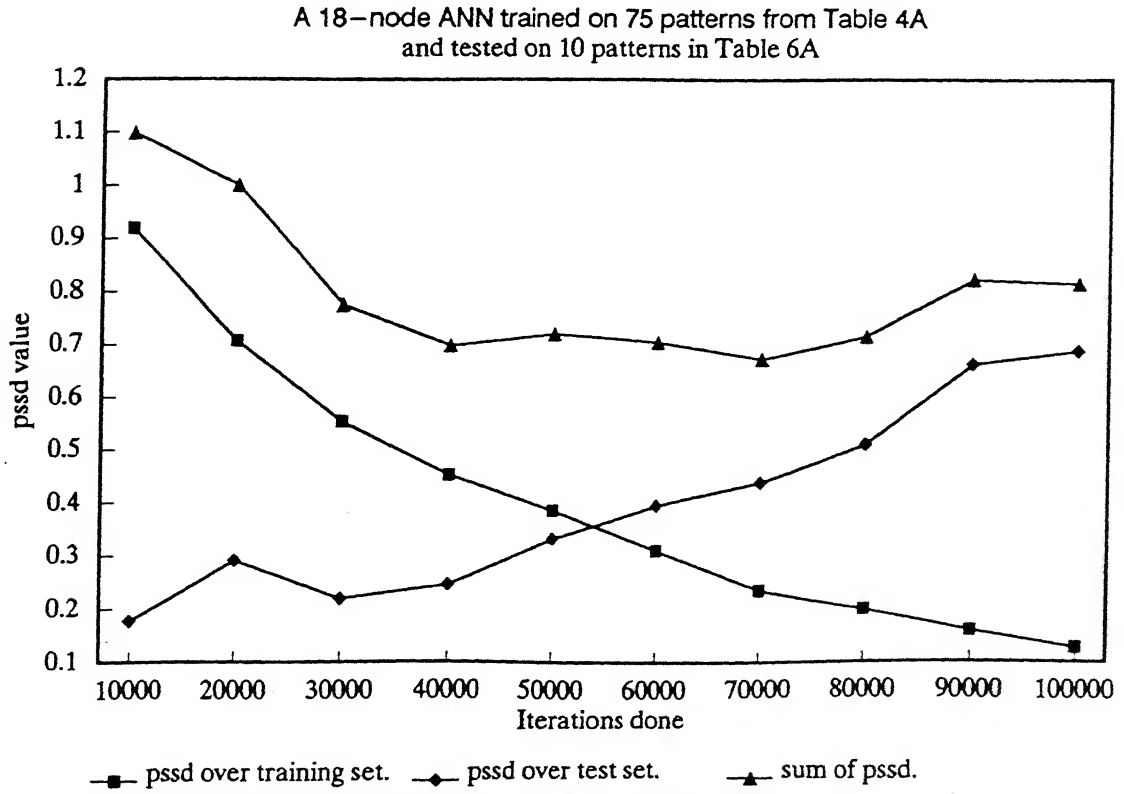


Fig. 6.19: Variation of cumulative pssd with iters for 18 nodes ANN.
The training set/test set had 75/10 patterns. Minima of 0.673832 at 70000 iters.

Experiment XIII.

Experiment with 18-node ANN trained on 100 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. (Table 6.19).

Table 6.19:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	1.90061	0.165253	2.065863
20000	1.21428	0.113507	1.327787
30000	0.910391	0.127668	1.038059
40000	0.830627	0.15355	0.984177
50000	0.77311	0.150179	0.923289
60000	0.724685	0.131218	0.855903
70000	0.694483	0.15036	0.844843
80000	0.662154	0.119429	0.781583
90000	0.646003	0.128156	0.774159
100000	0.63076	0.135009	0.765769

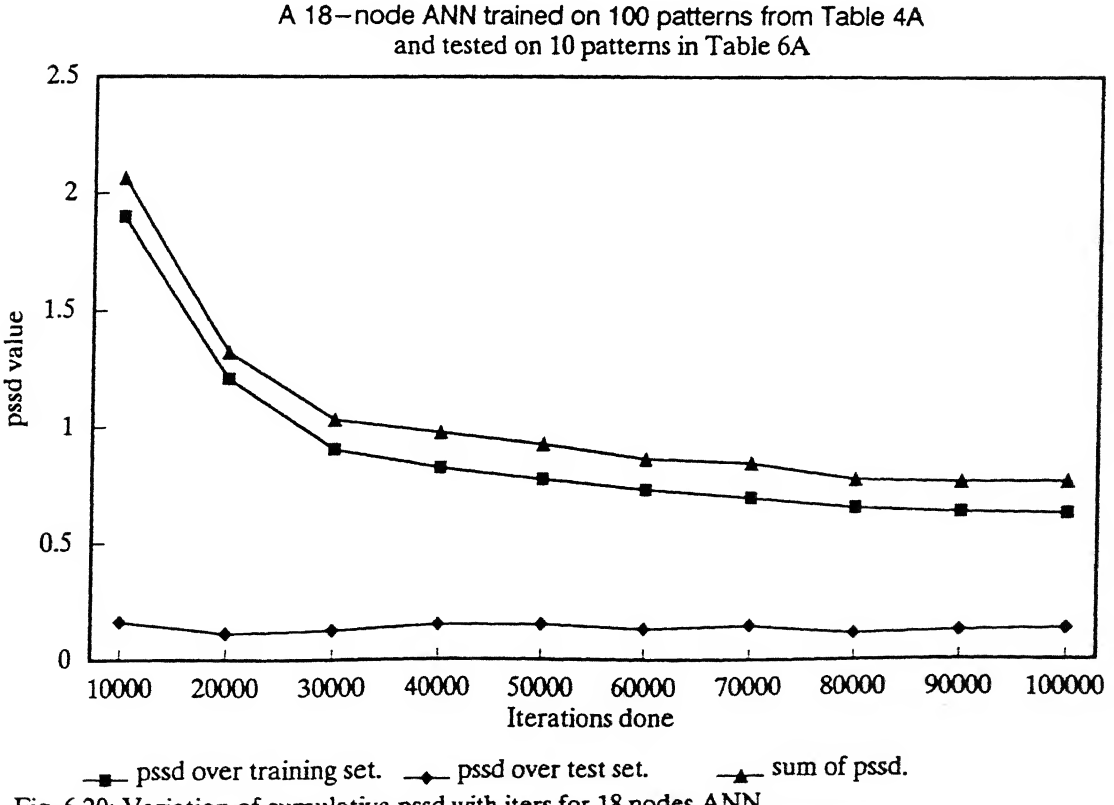


Fig. 6.20: Variation of cumulative pssd with iters for 18 nodes ANN.
The training set/test set had 100/10 patterns. Minima of 0.765769 at 100000 iters.

Experiment XIV.

Experiment with 6-node ANN trained on 50 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. See Table 6.20.

Table 6.20:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	1.00186	0.967746	1.969606
20000	0.817872	1.11509	1.932962
30000	0.772933	1.0649	1.837833
40000	0.727196	0.988202	1.715398
50000	0.705284	0.948153	1.653437
60000	0.697067	0.933641	1.630708
70000	0.694248	0.928691	1.622939
80000	0.693437	0.926155	1.619592
90000	0.693388	0.922635	1.616023
100000	0.693595	0.919759	1.613354

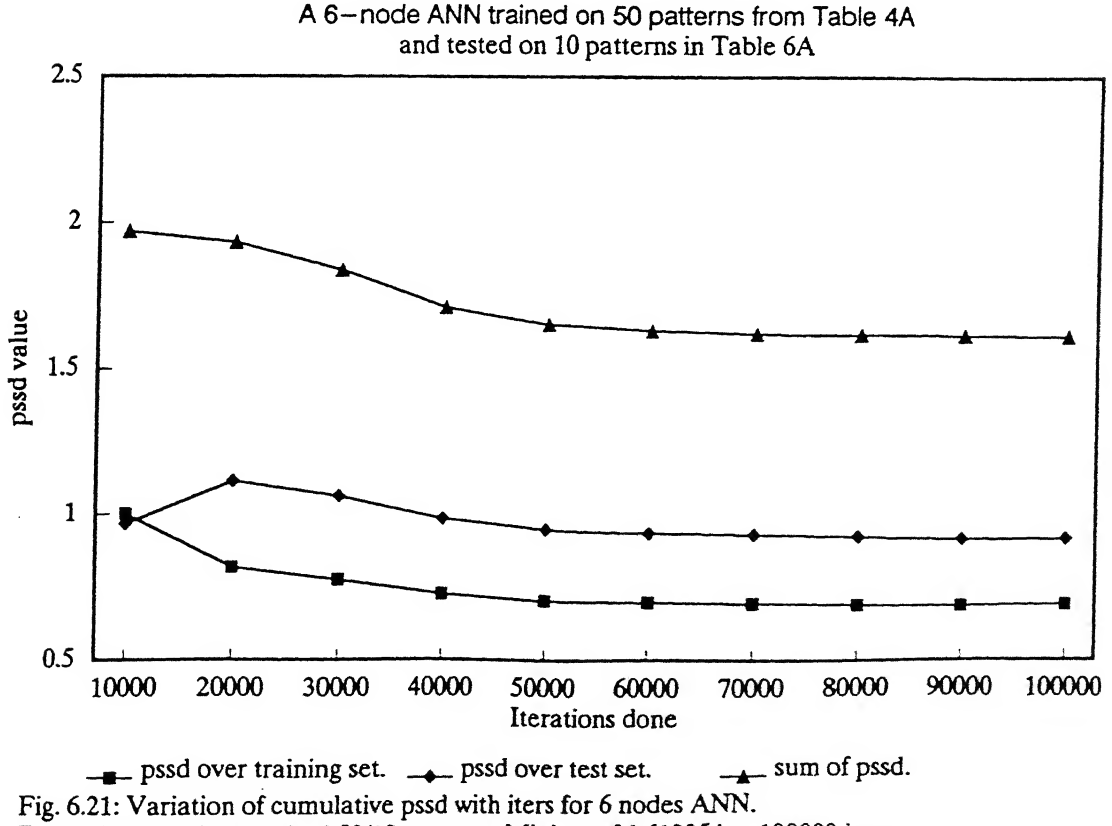


Fig. 6.21: Variation of cumulative pssd with iters for 6 nodes ANN.
The training set/test set had 50/10 patterns. Minima of 1.613354 at 100000 iters.

Experiment XV.

Experiment with 6-node ANN trained on 75 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. (Table 6.21).

Table 6.21:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	1.13076	0.357672	1.488432
20000	0.98433	0.2835	1.26783
30000	0.945936	0.291224	1.23716
40000	0.929682	0.2952	1.224882
50000	0.921618	0.29746	1.219078
60000	0.91767	0.299153	1.216823
70000	0.915957	0.300664	1.216621
80000	0.915246	0.30173	1.216976
90000	0.914969	0.302271	1.21724
100000	0.914861	0.302416	1.217277

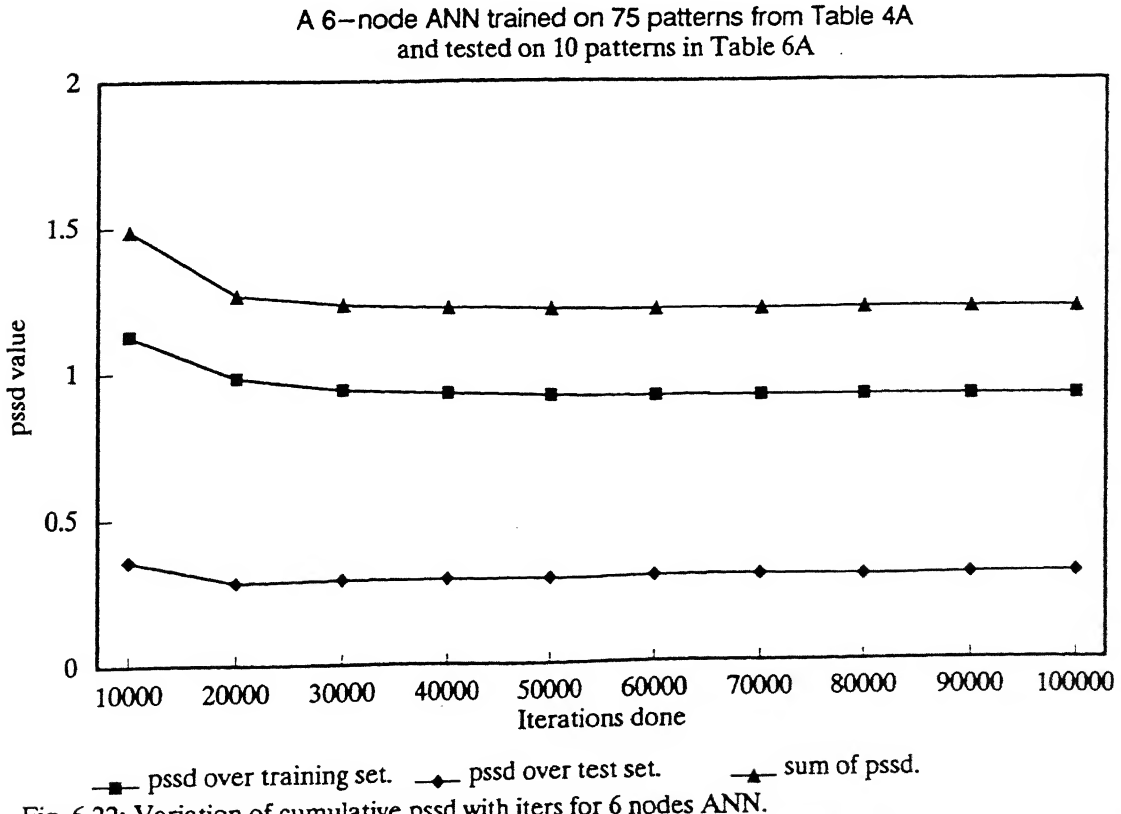


Fig. 6.22: Variation of cumulative pssd with iters for 6 nodes ANN.
The training set/test set had 75/10 patterns. Minima of 1.216621 at 70000 iters.

Experiment XVI.

Experiment with 6-node ANN trained on 100 patterns (from Table 4A) and its generalizing ability assessed over 10 patterns (in Table 6A) in the test set. (Table 6.22).

Table 6.22:

Iters done	pssd over training set	pssd on the test set	sum of pssds
10000	2.23855	0.304743	2.543293
20000	1.95866	0.283244	2.241904
30000	1.80277	0.249068	2.051838
40000	1.74494	0.222802	1.967742
50000	1.64723	0.199754	1.846984
60000	1.60712	0.211686	1.818806
70000	1.57032	0.241935	1.812255
80000	1.54798	0.250603	1.798583
90000	1.53691	0.25613	1.79304

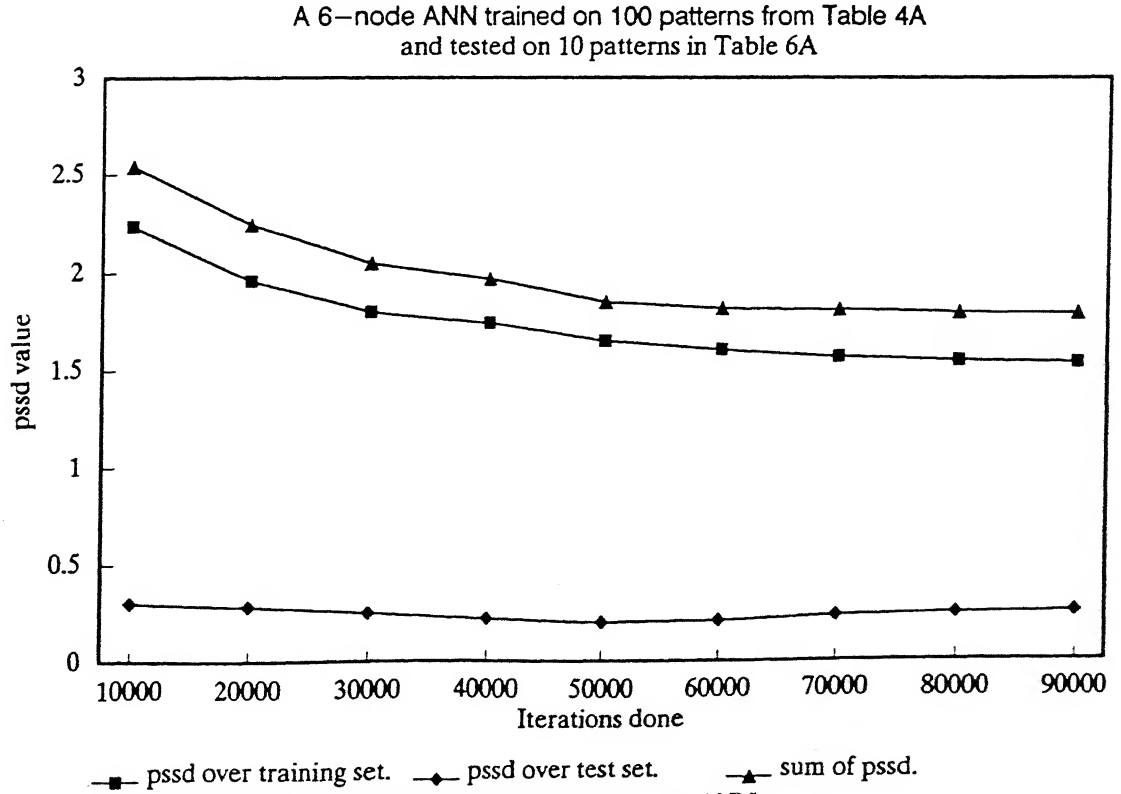


Fig. 6.23: Variation of cumulative pssd with iters for 6 nodes ANN.
The training set/test set had 100/10 patterns. Minima of 1.79304 at 90000 iters.

Remarks.

Several aspects of the above experiments are noteworthy. In particular,

* The overall pssd, obtained by the summation of the pssd on the training set *and* the test set, shows a smoother behavior. It sometimes reaches the *bottom* of a trough (Fig.6.15, 6.16, 6.17, 6.18, 6.19 etc.) that we are ardently hunting for.

* We started experimenting with 50 patterns on nets and subsequently increased this number to 75 and then to 100. Somewhat surprisingly all the nets(6,12,18) showed the smallest pssd when trained with 75 patterns. The performance deteriorated (i.e., pssd increased) when the number of training patterns were increased beyond 75.

* Since it has been suggested by Lippman *(1987)* that the number of nodes in the intermediate layer must always be more than three times the number of nodes in the preceding layer we started our experiments with 12 nodes ANN (as there were 4 nodes in our input layer). The best pssd was 0.745039. Next, we thought of making trials with 18 nodes and found the best pssd value of 0.673832. The best pssd with 6 nodes ANN was 1.216621.

Thus, one can infer that with even larger number of nodes (greater than 18 here) chosen and training this ANN on these 75 patterns we might end up with better predictive capability at least for the present application. Clearly the word of caution here would be that these inferences are probably problem dependent and whatever may be said with regard to one problem domain may not be necessarily true for others. This conclusion appears to be consistent with earlier reported applications of ANNs that attempt to *generalize* relationships (rather than classify).

CHAPTER 7
CONCLUSIONS AND DIRECTIONS FOR FURTHER RESEARCH

Chapter 6 records our experience as we attempted to reach the optimum ANN configuration empirically because no formal guidelines appear to be available as to what the optimum values for *learning rate (η), momentum factor (α), tolerance (ε), the number of patterns required for training (P) and the ANN's architecture (viz., the number of hidden layers and the number of nodes in them)* should be. The experiments do show that the ANN performance is significantly affected by the choice of these parameters. However, even the "best" parameter values identified leave considerable amount of residual error in the predictions as evidenced from the outputs of the trained nets. Table 7.1 shows an output from a 12-node ANN which was trained on 50 patterns in accordance to the cross validation technique (secn. in 6.4) with learning rate of 0.9 and momentum factor 0.1. The resulting total pattern sum of squared deviation (pssd) over the test set in this case was 0.387049. Table 7.2 is an output of the same net when used for predicting sample points in Table 3A (see appendix). The pssd in this case was 0.030461.

Table 7.1:

targ_n	pred_n	err_n	perr_n	targ_c	pred_c	err_c	perr_c
24.0	27.7	3.7	15.31	2.0	2.3	0.3	13.40
7.0	6.8	-0.2	3.46	5.0	4.9	-0.1	2.81
44.0	43.2	-0.8	1.82	5.0	3.3	-1.7	33.70
23.0	17.1	-5.9	25.80	3.0	2.2	-0.8	26.69
44.0	40.8	-3.2	7.29	3.0	2.9	-0.1	3.18
26.0	15.8	-10.2	39.12	5.0	2.8	-2.2	43.27
22.0	30.7	8.7	39.41	4.0	4.9	0.9	23.49
17.0	21.0	4.0	23.48	3.0	3.9	0.9	28.40
24.0	20.4	-3.6	14.99	4.0	3.5	-0.5	12.56
23.0	28.9	5.9	25.53	1.0	1.6	0.6	59.11

Table 7.2

targ_n	pred_n	err_n	perr_n	targ_c	pred_c	err_c	perr_c
12.0	9.4	-2.6	21.33	5.0	4.6	-0.4	8.95
34.0	33.5	-0.5	1.35	4.0	4.3	0.3	6.58
29.0	26.5	-2.5	8.49	3.0	2.9	-0.1	4.57
10.0	10.8	0.8	7.95	4.0	4.5	0.5	12.37
42.0	43.5	1.5	3.45	4.0	4.3	0.3	8.70
7.0	8.2	1.2	17.73	5.0	5.1	0.1	2.12
39.0	41.2	2.2	5.57	1.0	1.0	0.0	0.11
7.0	8.5	1.5	20.91	5.0	5.1	0.1	2.14
17.0	17.8	0.8	4.94	1.0	1.2	0.2	17.73
20.0	21.5	1.5	7.37	4.0	4.3	0.3	7.07

Table 7.3 is an output of a 18-node ANN, having minimum cumulative pssd in Study Three (see secn 6.5), trained on 75 and tested over 10 patterns in Tables 4A and 6A respectively (shown in appendix). Table 7.4 lists the outcome of prediction from a 12-node ANN adjudged best among the nets trained and tested over sets having an equal number of patterns (50 each) as given in Tables 4A and 5A respectively (see appendix).

Table 7.3:

targ_n	pred_n	err_n	perr_n	targ_c	pred_c	err_c	perr_c
41.0	37.36	-3.64	8.89	4.0	2.40	-1.60	39.92
47.0	50.29	3.29	7.00	3.0	4.76	1.76	58.78
27.0	26.26	-0.74	2.76	5.0	4.39	-0.61	12.21
43.0	43.49	0.49	1.15	2.0	1.87	-0.13	6.46
12.0	10.00	-2.00	16.69	5.0	3.62	-1.38	27.59
41.0	40.67	-0.33	0.80	0.0	-0.17	-0.17	+INF
10.0	7.68	-2.32	23.20	1.0	-0.13	-1.13	112.55
43.0	41.47	-1.53	3.56	5.0	3.79	-1.21	24.16
11.0	6.72	-4.28	38.90	3.0	1.13	-1.87	62.32
40.0	32.00	-8.00	20.01	4.0	2.98	-1.02	25.56

Table 7.4:

targ_n	pred_n	err_n	perr_n	targ_c	pred_c	err_c	perr_c
34.0	37.57	3.57	10.51	4.0	5.33	1.33	33.13
29.0	37.33	8.33	28.74	3.0	5.37	2.37	79.15
10.0	8.84	-1.16	11.60	4.0	5.05	1.05	26.15
42.0	42.30	0.30	0.71	4.0	4.74	0.74	18.39
13.0	14.90	1.90	14.59	3.0	3.86	0.86	28.82
39.0	29.71	-9.29	23.82	1.0	0.12	-0.88	88.13
43.0	49.12	6.12	14.23	0.0	-0.06	-0.06	+INF
17.0	6.23	-10.77	63.36	0.0	-0.62	-0.62	+INF
20.0	12.24	-7.76	38.82	4.0	2.90	-1.10	27.60
23.0	20.11	-2.89	12.58	4.0	3.92	-0.08	2.01
38.0	33.71	-4.29	11.28	3.0	2.96	-0.04	1.32
43.0	38.81	-4.19	9.75	4.0	3.77	-0.23	5.83
16.0	15.14	-0.86	5.38	3.0	3.58	0.58	19.18
20.0	18.60	-1.40	7.00	5.0	4.17	-0.83	16.58
33.0	23.17	-9.83	29.79	1.0	0.33	-0.67	67.31
42.0	48.47	6.47	15.41	3.0	3.78	0.78	25.90
49.0	45.94	-3.06	6.24	1.0	0.23	-0.77	77.24
37.0	29.14	-7.86	21.26	4.0	3.79	-0.21	5.14
27.0	17.98	-9.02	33.42	4.0	3.24	-0.76	18.92
16.0	10.93	-5.07	31.66	0.0	-0.01	-0.01	+INF
20.0	8.92	-11.08	55.40	1.0	-0.47	-1.47	146.67
44.0	34.11	-9.89	22.47	0.0	-0.58	-0.58	+INF
34.0	37.52	3.52	10.35	3.0	4.73	1.73	57.75
43.0	48.50	5.50	12.78	3.0	3.66	0.66	21.98
13.0	16.93	3.93	30.20	4.0	3.62	-0.38	9.46
17.0	16.26	-0.74	4.33	3.0	2.95	-0.05	1.61
39.0	14.53	-24.47	62.75	1.0	-0.49	-1.49	148.97
21.0	29.91	8.91	42.42	0.0	0.50	0.50	+INF
46.0	40.90	-5.10	11.08	0.0	-0.36	-0.36	+INF
41.0	40.74	-0.26	0.64	2.0	2.99	0.99	49.62
34.0	26.24	-7.76	22.83	4.0	3.95	-0.05	1.21
14.0	16.33	2.33	16.65	1.0	-0.02	-1.02	102.13
31.0	11.77	-19.23	62.04	3.0	-0.10	-3.10	103.47
33.0	23.51	-9.49	28.76	4.0	3.70	-0.30	7.60
18.0	13.91	-4.09	22.71	0.0	-0.10	-0.10	+INF
47.0	37.66	-9.34	19.88	5.0	2.38	-2.62	52.49
39.0	15.22	-23.78	60.96	1.0	0.32	-0.68	67.76
39.0	29.76	-9.24	23.70	5.0	1.90	-3.10	62.07
33.0	34.06	1.06	3.20	2.0	3.22	1.22	60.81
14.0	19.38	5.38	38.40	4.0	4.60	0.60	15.02
23.0	26.90	3.90	16.97	1.0	2.29	1.29	129.37
43.0	28.45	-14.55	33.85	4.0	2.05	-1.95	48.84
14.0	17.71	3.71	26.53	5.0	3.91	-1.09	21.77
23.0	20.12	-2.88	12.53	0.0	0.13	0.13	+INF
19.0	9.00	-10.00	52.64	1.0	0.00	-1.00	100.42
35.0	33.71	-1.29	3.68	5.0	4.39	-0.61	12.15
41.0	18.66	-22.34	54.49	0.0	-0.51	-0.51	+INF
11.0	19.03	8.03	72.97	4.0	4.02	0.02	0.61
30.0	27.44	-2.56	8.53	2.0	3.00	1.00	50.21
42.0	26.85	-15.15	36.06	3.0	0.65	-2.35	78.39

A further attempt was made to reduce the errors in predictions by making use of the cross validation technique combined with lowered values of the momentum factor. The overall response remained almost the same as earlier while the predictive capability improved, albeit marginally (see Table 7.2). In the present work predictions of n and c given α, AQL, β, RQL ranged to $\pm$5 to 25%.

Table 7.5: Comparative performance of a 12-node ANN with various settings of the learning rate and momentum factor employed.

Learning rate(η)	Momentum factor(α)	pssd on P1	pssd on P2
0.6	0.5	0.441796	0.036145
0.9	0.3	0.402755	0.035967
0.9	0.1	0.387049	0.030461

In going from 50 and to 75 training patterns we observed consistent improvement in the ANN's performance as evidenced from Table 7.3.

Table 7.6: Effect of increase in number of training patterns on ANN performance.

No. of nodes	Cumulative pssd function of No. of Training patts		
	50	75	100
6	1.613354	1.216621	1.793040
12	1.348080	0.745039	1.085486
18	2.348758	0.673832	0.765769

It should be recalled that in the present work we are trying to simulate an exact mathematical relation given by

$$P(m<=c) = \sum_{m=0}^{c} \frac{n!}{(n-m)!m!} p^m (1-p)^{n-m}$$

where P is the probability of acceptance of the lot,

p is the fraction defective in the lot,

n is the sample size, and

c the acceptance number

for the range $(10<n<50)$, $(0<c<5)$ and $(0.01<p<0.25)$. Clearly

presenting all possible representative input values in this domain to the ANN would require significantly large number of (perhaps several hundred) patterns. As this work was merely an evaluation of the ANN approach to design of single sampling plan schemes, we stopped with 100 patterns, rather than consuming excessive computing time.

The feasibility of using ANNs to design single sampling plans has been demonstrated. However, further work to optimize the ANN parameters appear to be a must if one is interested in delivering the ANN substitute for the Larson nomograph. We are not content with the results the ANN paradigm has delivered using upto 100 patterns. We would therefore, strongly recommend use of perhaps 500 or more patterns (with entropy carefully maximized). However, most certainly that would require use of faster machines than the HP9000/850.

REFERENCES

(1) Bagchi T.P., *Taguchi Methods Explained: Practical steps to Robust Design*, Prentice Hall, New Delhi, 1993.

(2) Baum E.B. and D. Haussler, ''What size net gives valid generalization ?,'' *Neural Computation*, 1: 151-160, 1989.

(3) Becker S. and Y. LeCun, ''Improving the Convergence of back-propagation learning with second order methods,'' In *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kauffmann, pp. 29-37, 1988.

(4) Bhagat P., ''An introduction to neural nets,'' *Chemical Engng. Progress*, pp. 55-60, Aug 1990.

(5) Blum A. and R.I. Rivest, ''Training a 3-node neural network is NP-complete,'' In *Proceedings of the Computational Learning Theory (COLT) Conference*, Morgan Kaufmann, pp. 9-18, 1988.

(6) Cottrel G.W. and F. Tsung, ''Learning simple arithmetic procedures,'' *Connection Science*, Vol. 5, No. 1, 1993.

(7) Cruz C.A., ''Understanding neural networks,'' *Graeme Publishing Corp.*, Amherst, NH, 1988.

(8) Cybenko G., ''Approximation by superpositions of a sigmoidal functio,'' *Mathematics of Control, Signals, and Systems*, 2(4): 303-314, 1989.

(9) Duncan A.J., *Quality Control and Industrial Statistics*, Richard D. Irwin Inc., Homewood, 1974.

(10) Grant E.L. and R.S. Leavenworth, *Statistical Quality Control*, 6th ed., McGraw Hill, 1988.

(11) Hammerstrom D., ''Neural networks at work,'' *IEEE Spectrum*, pp. 26-32, June 1993.

(12) Hart A., ''Using neural networks for classification tasks - some experiments on datasets and practical advice,'' *Jornal of Operational Research Society*, Vol. 43, No. 3, pp. 215-226, 1992.

(13) Hebb D.O., *The Organization of Behavior*, John Wiley & Sons, New York, 1949.

(14) Hinton G.E., ''Connectionist leearning procedures,''

Technical Report CMU CS-86-126, Carnegie - Mellon University, Pittsburgh, PA, 1986.

(15) Hoskins J.C. and D.M. Himmelblau, ''Artificial neural network models of knowledge representation in Chemical Engineering,'' *Computers Chem. Engng.*, Vol. 12, No. 9/10, pp. 881-890, 1988.

(16) Hush D.R. and J.M. Salas, ''Improving the learning rate of back-propagation with the gradient reuse algorithm,'' In *Proceedings of the IEEE International Conference on Neural Networks*, Vol. 1, pp. 441-448, 1988.

(17) Hush D., J. Salas, and B. Horne, ''Error surface for multilayer perceptrons,'' *International joint Conference on Neural Networks*, Seattle, Washington, July 8-12, I, pp. 759-764, 1991.

(18) Hush D.R. and B.G.Horne, ''Progress in supervised neural networks,'' *IEEE Signal Processing Magazine*, pp. 8-39, Jan 1993.

(19) Huang S.C. and Y.F. Huang, ''Bounds on the number of hidden neurons in multilayer perceptrons,'' *IEEE Transactions on Neural Networks*, 2(1): 47-55, 1991.

(20) Hwarng H.B. and N.F. Hubele, ''$\bar{X}$ control chart pattern identification through efficient off-line neural network training,'' *IIE Transactions*, Vol. 25, No. 3, pp. 27-40, 1993.

(21) Hwarng H.B. and N.F. Hubele, ''Backpropagation pattern recognizers for $\bar{X}$ control charts: Methodology and Performance,'' *Computers 'Indl. Engng.*, Vol. 24, No. 2, pp. 219-235, 1993.

(22) Jacobs R.A., ''Increased rates of convergence through learning rate adaptation,'' *Neural Networks*, 1(4): 295-308, 1988.

(23) Judd J.S., *Neural Network Design and the Complexity of Learning*, MIT Press, Cambridge, MA, 1990.

(24) Juran J.M. and F.M. Gryna, *Juran's Quality Control Handbook*, 4th ed., McGraw Hill, 1988.

(25) Kohonen T., *Self-Organization and Associative memory*, Springer-Verlag, Berlin, 1984.

(26) LeCun Y., J.S. Denker and S.A. Solla, ''Optimal brain damage,'' In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, Morgan Kaufmann, pp. 598-605, 1990.

(27) Lee Y., S.Oh, and M. Kim, ''The effect of initial weights on premature saturation in back-propagation learning,''

International Joint Conference on Neural Networks, Seattle, Washington, July 8-12, I, pp. 765-770, 1991.

(28) Leonard J.A. and M.A. Kramer, ''Diagnosis using back-propagation neural networks - Analysis and Criticism,'' *Computers Chem. Engng.*, Vol. 14, No. 12, pp. 1323-1338, 1990.

(29) Lippmann R.P., ''An introduction to computing with neural nets,'' *IEEE Acoustics, Speech and Signal Processing Magazine*, 4(2): 4-22, April 1987.

(30) Makhoul J., A. EI-Jaroudi, and Schwartz, ''Formation of disconnected decision regions with a single hidden layer,'' In *Proceedings of the International Joint Conference on Neural Networks*, Vol. 1, pp. 455-460, 1989.

(31) Phadke M.S., *Quality Engineering using Robust Design*, Prentice Hall, 1989.

(32) Plaut D.C., S.J. Nowlan, and G.E. Hinton, ''Experiments on learning back-propagation,'' *Technical Report CMU CS-86-126*, Carnegie - Mellon University, Pittsburgh, PA, 1986.

(33) Quantrille T.E. and Y.A. Liu, *Artificial Intelligence in Chemical Engineering*, Academic Press Incl., 1991.

(34) Rosenblatt R., *Principles of Neurodynamics*, New York, Spartan Books, 1959.

(35) Rumelhart D.E., G.E. Hinton and R.J. Williams, ''Learning representations by backpropagating errors,'' *Nature*, Vol. 323, pp. 533-536, Oct 1986.

(36) Rumelhart D.E. and J.L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, Vol. 1, MIT Press, Cambridge, MA, 1986.

(37) Rumelhart D.E. and J.L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, Vol. 2, MIT Press, Cambridge, MA, 1986.

(38) Schilling E.G., *Acceptance Sampling in Quality Control*, Marcel Dekker Inc., NY, 1982.

(39) Simpson P.K., *Artificial Neural Systems*, Pergamon Press, Elmsford, NY, 1989.

(40) Smith A.E. and C.H. Dagli, ''Controlling industrial processes through supervised feedforward neural networks,'' *Computers and Indl. Engng.*, 1991.

(41) Tesauro G., ''Scaling relationships in back-propagation learning: dependence on training set size,'' *Complex Systems*, Vol. 1, pp. 367-372, 1987.

(42) Venkatasubramanian V., R. Vaidyanathan and Y. Yamamoto, ''Process fault detection and diagnosis using neural networks-I. Steady state processes,'' *Computers Chem. Engng.*, Vol. 14, No. 7, pp. 699-712, 1990.

(43) Wassermann P.D., *Neural Computing: Theory and Practice*, Van Nostrand Reinhold, New York, NY, 1989.

(44) Watrous R.L., ''Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimization,'' In *proceedings IEEE 1st International Conference on Neural Networks*, Vol. 2, pp. 619-628, 1987.

APPENDIX A

Table 1A: Training Set 1.

0.082012 0.915544 0.162626 0.479863 29 5
5.1785714286E-01 8.3333333333E-01
0.085890 0.106019 0.197707 0.000893 43 1
7.6785714286E-01 1.6666666667E-01
0.093644 0.998958 0.161234 0.983691 14 5
2.5000000000E-01 8.3333333333E-01
0.080352 0.781945 0.106532 0.601112 30 3
5.3571428571E-01 5.0000000000E-01
0.093493 0.160128 0.193917 0.006022 34 1
6.0714285714E-01 1.6666666667E-01
0.053881 0.992864 0.158222 0.880582 8 2
1.4285714286E-01 3.3333333333E-01
0.071564 0.974668 0.180377 0.558792 24 4
4.2857142857E-01 6.6666666667E-01
0.097211 0.854515 0.148706 0.563361 29 4
5.1785714286E-01 6.6666666667E-01
0.039381 0.997686 0.184395 0.705925 15 3
2.6785714286E-01 5.0000000000E-01
0.012273 0.999123 0.224511 0.030007 35 3
6.2500000000E-01 5.0000000000E-01
0.038945 0.954497 0.191463 0.462394 9 1
1.6071428571E-01 1.6666666667E-01
0.095225 0.725845 0.236195 0.042591 37 5
6.6071428571E-01 8.3333333333E-01
0.091794 0.999996 0.106069 0.999991 7 5
1.2500000000E-01 8.3333333333E-01
0.022682 0.999822 0.106853 0.426754 24 4
4.2857142857E-01 6.6666666667E-01
0.061121 0.880168 0.181724 0.268300 20 2
3.5714285714E-01 3.3333333333E-01
0.062738 0.716873 0.118580 0.233668 43 5
7.6785714286E-01 8.3333333333E-01
0.064164 0.473198 0.180975 0.010207 43 2
7.6785714286E-01 3.3333333333E-01
0.045587 0.995959 0.221068 0.361502 24 4
4.2857142857E-01 6.6666666667E-01
0.024797 0.976645 0.229665 0.004638 44 3
7.8571428571E-01 5.0000000000E-01
0.021145 0.959788 0.203257 0.014342 36 2
6.4285714286E-01 3.3333333333E-01
0.025777 0.992543 0.161153 0.511069 16 2
2.8571428571E-01 3.3333333333E-01
0.068548 0.738168 0.242090 0.009583 38 3
6.7857142857E-01 5.0000000000E-01
0.075837 0.929761 0.240202 0.362951 13 2
2.3214285714E-01 3.3333333333E-01
0.042975 0.371909 0.233250 0.000035 49 1
8.7500000000E-01 1.6666666667E-01
0.024892 0.718846 0.169912 0.003849 42 1
7.5000000000E-01 1.6666666667E-01
0.034480 0.926501 0.109053 0.246569 46 5
8.2142857143E-01 8.3333333333E-01
0.072737 0.999092 0.176478 0.935945 17 5
3.0357142857E-01 8.3333333333E-01
0.036423 0.999800 0.184146 0.843328 16 4
2.8571428571E-01 6.6666666667E-01
0.092783 0.858562 0.121404 0.715680 22 3
3.9285714286E-01 5.0000000000E-01

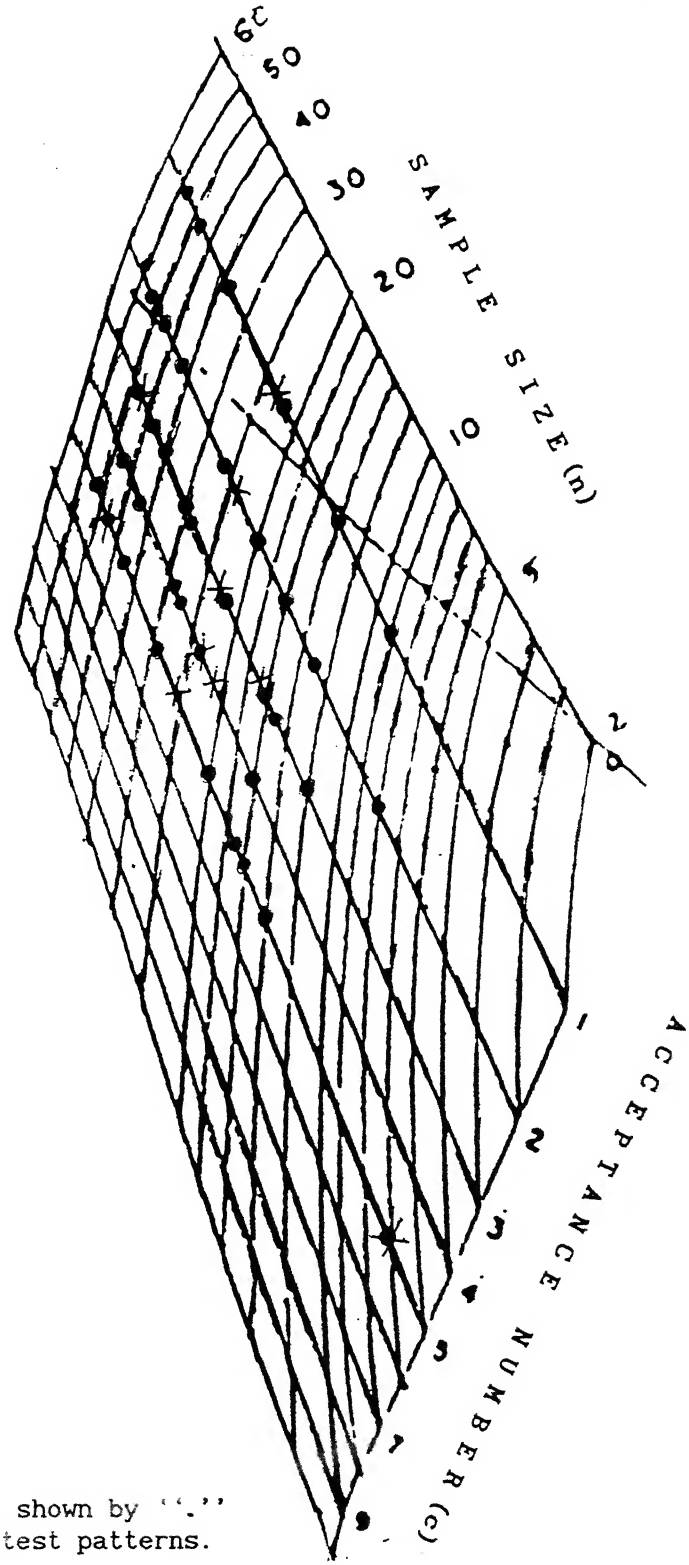
Table 1A contd...

0.010217 0.988326 0.124045 0.063898 46 2
8.2142857143E-01 3.3333333333E-01
0.064946 0.770383 0.149254 0.359577 14 1
2.5000000000E-01 1.6666666667E-01
0.011460 0.999992 0.196900 0.802741 12 3
2.1428571429E-01 5.0000000000E-01
0.083335 0.991973 0.129688 0.953320 16 4
2.8571428571E-01 6.6666666667E-01
0.033732 0.943990 0.217237 0.057035 26 2
4.6428571429E-01 3.3333333333E-01
0.099387 0.675227 0.176995 0.219571 29 3
5.1785714286E-01 5.0000000000E-01
0.061745 0.499535 0.242164 0.000709 43 2
7.6785714286E-01 3.3333333333E-01
0.065704 0.672569 0.115620 0.235721 44 3
7.8571428571E-01 5.0000000000E-01
0.034080 0.998234 0.118819 0.886971 16 3
2.8571428571E-01 5.0000000000E-01
0.056248 0.646713 0.112502 0.274274 22 1
3.9285714286E-01 1.6666666667E-01
0.073427 0.348604 0.190487 0.004924 45 2
8.0357142857E-01 3.3333333333E-01
0.029067 0.975989 0.195423 0.044522 38 3
6.7857142857E-01 5.0000000000E-01
0.033447 0.945013 0.150933 0.093728 43 4
7.6785714286E-01 6.6666666667E-01
0.067777 0.980141 0.245585 0.421035 16 3
2.8571428571E-01 5.0000000000E-01
0.071050 0.997914 0.127680 0.965803 20 5
3.5714285714E-01 8.3333333333E-01
0.091794 0.999996 0.106069 0.999991 7 5
1.2500000000E-01 8.3333333333E-01
0.088011 0.488186 0.248671 0.003202 42 3
7.5000000000E-01 5.0000000000E-01
0.072163 0.122560 0.159412 0.002075 49 1
8.7500000000E-01 1.6666666667E-01
0.045607 0.974505 0.182439 0.169596 37 4
6.6071428571E-01 6.6666666667E-01
0.082242 0.933743 0.133168 0.713380 27 4
4.8214285714E-01 6.6666666667E-01
0.076854 0.999161 0.176997 0.950920 16 5
2.8571428571E-01 8.3333333333E-01

Table 2A: Prediction Set 1.

```
0.020803 0.986855 0.246993 0.042598   24    2
4.2857142857E-01 3.3333333333E-01
0.076914 0.999999 0.158002 0.999906    7    5
1.2500000000E-01 8.3333333333E-01
0.061917 0.710785 0.205909 0.012060   44    5
7.8571428571E-01 8.3333333333E-01
0.032161 0.994197 0.101044 0.802252   23    3
4.1071428571E-01 5.0000000000E-01
0.074750 0.580343 0.221749 0.006423   44    3
7.8571428571E-01 5.0000000000E-01
0.017273 0.999995 0.148842 0.819708   26    5
4.6428571429E-01 8.3333333333E-01
0.095654 0.946973 0.217181 0.462729   22    4
3.9285714286E-01 6.6666666667E-01
0.035150 0.997485 0.219630 0.468161   17    3
3.0357142857E-01 5.0000000000E-01
0.036373 0.998485 0.240158 0.282771   24    4
4.2857142857E-01 6.6666666667E-01
0.035471 0.804352 0.144766 0.134137   23    1
4.1071428571E-01 1.6666666667E-01
```

Fig. 1A: Nomograph with patterns in Table 1A and 2A.

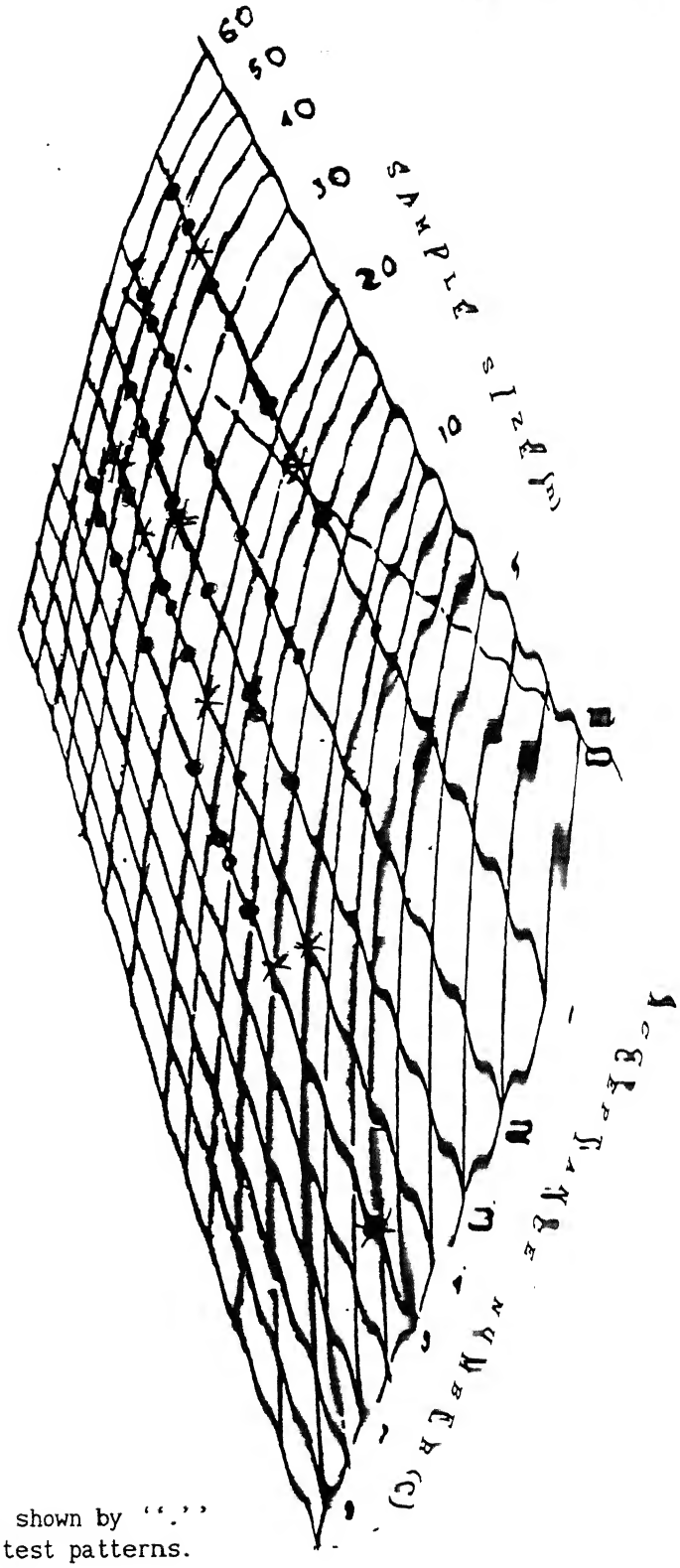


The training patterns are shown by ''.'' and, ''**x**'' represent the test patterns.

Table 3A: Prediction Set 2.

0.089720	0.999701	0.233808	0.959723	12	5
2.1428571429E-01	8.3333333333E-01				
0.061539	0.944619	0.143413	0.450240	34	4
6.0714285714E-01	6.6666666667E-01				
0.036066	0.980446	0.129609	0.471078	29	3
5.1785714286E-01	5.0000000000E-01				
0.094592	0.998731	0.237850	0.935265	10	4
1.7857142857E-01	6.6666666667E-01				
0.054331	0.807188	0.123620	0.220503	42	4
7.5000000000E-01	6.6666666667E-01				
0.095846	0.999995	0.203684	0.999587	7	5
1.2500000000E-01	8.3333333333E-01				
0.035195	0.599010	0.172688	0.005624	39	1
6.9642857143E-01	1.6666666667E-01				
0.086932	0.999997	0.106787	0.999991	7	5
1.2500000000E-01	8.3333333333E-01				
0.012025	0.982557	0.198177	0.121752	17	1
3.0357142857E-01	1.6666666667E-01				
0.040132	0.999027	0.243348	0.442101	20	4
3.5714285714E-01	6.6666666667E-01				

Fig. 2A: Nomograph with patterns in Table 1A and 3. A.



The training patterns are shown by ''.'' and, ''**x**'' represent the test patterns.

Table 4A: Training Pattern Set 2.

0.082012 0.915544 0.162626 0.479863 29 5
4.8000000000E-01 9.0000000000E-01
0.085890 0.106019 0.197707 0.000893 43 1
7.6000000000E-01 2.6000000000E-01
0.093644 0.252456 0.161234 0.085304 14 0
1.8000000000E-01 1.0000000000E-01
0.080352 0.781945 0.106532 0.601112 30 3
5.0000000000E-01 5.8000000000E-01
0.093493 0.035532 0.193917 0.000656 34 0
5.8000000000E-01 1.0000000000E-01
0.053881 0.212071 0.158222 0.008045 28 0
4.6000000000E-01 1.0000000000E-01
0.054471 0.260734 0.101934 0.075752 24 0
3.8000000000E-01 1.0000000000E-01
0.097211 0.041824 0.148706 0.003584 49 1
8.8000000000E-01 2.6000000000E-01
0.039381 0.997686 0.184395 0.705925 15 3
2.0000000000E-01 5.8000000000E-01
0.012273 0.999123 0.224511 0.030007 35 3
6.0000000000E-01 5.8000000000E-01
0.020803 0.986855 0.246993 0.042598 24 2
3.8000000000E-01 4.2000000000E-01
0.076914 0.995914 0.158002 0.898898 21 5
3.2000000000E-01 9.0000000000E-01
0.061917 0.710785 0.205909 0.012060 44 5
7.8000000000E-01 9.0000000000E-01
0.032161 0.471484 0.101044 0.086295 23 0
3.6000000000E-01 1.0000000000E-01
0.074750 0.580343 0.221749 0.006423 44 3
7.8000000000E-01 5.8000000000E-01
0.017273 0.999995 0.148842 0.819708 26 5
4.2000000000E-01 9.0000000000E-01
0.095654 0.946973 0.217181 0.462729 22 4
3.4000000000E-01 7.4000000000E-01
0.035150 0.997485 0.219630 0.468161 17 3
2.4000000000E-01 5.8000000000E-01
0.036373 0.998485 0.240158 0.282771 24 4
3.8000000000E-01 7.4000000000E-01
0.035471 0.804352 0.144766 0.134137 23 1
3.6000000000E-01 2.6000000000E-01
0.038945 0.982680 0.191463 0.238598 26 3
4.2000000000E-01 5.8000000000E-01
0.095225 0.725845 0.236195 0.042591 37 5
6.4000000000E-01 9.0000000000E-01
0.022682 0.999822 0.206853 0.426754 24 4
3.8000000000E-01 7.4000000000E-01
0.061121 0.880168 0.181724 0.268300 20 2
3.0000000000E-01 4.2000000000E-01
0.062738 0.716873 0.118580 0.233668 43 5
7.6000000000E-01 9.0000000000E-01
0.064164 0.473198 0.180975 0.010207 43 2
7.6000000000E-01 4.2000000000E-01
0.045587 0.995959 0.221068 0.361502 24 4
3.8000000000E-01 7.4000000000E-01
0.024797 0.976645 0.229665 0.004638 44 3
7.8000000000E-01 5.8000000000E-01
0.021145 0.959788 0.203257 0.014342 36 2
6.2000000000E-01 4.2000000000E-01
0.025777 0.658471 0.161153 0.060107 16 0

Table 4A contd...

2.2000000000E-01 1.0000000000E-01
0.068548 0.738168 0.242090 0.009583 38 3
6.6000000000E-01 5.8000000000E-01
0.075837 0.929761 0.240202 0.362951 13 2
1.6000000000E-01 4.2000000000E-01
0.042975 0.371909 0.233250 0.000035 49 1
8.8000000000E-01 2.6000000000E-01
0.024892 0.718846 0.169912 0.003849 42 1
7.4000000000E-01 2.6000000000E-01
0.034480 0.926501 0.109053 0.246569 46 5
8.2000000000E-01 9.0000000000E-01
0.072737 0.276979 0.176478 0.036853 17 0
2.4000000000E-01 1.0000000000E-01
0.036423 0.999800 0.184146 0.843328 16 4
2.2000000000E-01 7.4000000000E-01
0.092783 0.858562 0.121404 0.725680 22 3
3.4000000000E-01 5.8000000000E-01
0.010217 0.988326 0.124045 0.063898 46 2
8.2000000000E-01 4.2000000000E-01
0.064946 0.770383 0.149254 0.359577 14 1
1.8000000000E-01 2.6000000000E-01
0.011460 0.999992 0.196900 0.802741 12 3
1.4000000000E-01 5.8000000000E-01
0.083335 0.248527 0.129688 0.108343 16 0
2.2000000000E-01 1.0000000000E-01
0.033732 0.409768 0.217237 0.001715 26 0
4.2000000000E-01 1.0000000000E-01
0.099387 0.675227 0.176995 0.219571 29 3
4.8000000000E-01 5.8000000000E-01
0.061745 0.499535 0.242164 0.000709 43 2
7.6000000000E-01 4.2000000000E-01
0.065796 0.050053 0.200344 0.000053 44 0
7.8000000000E-01 1.0000000000E-01
0.065704 0.413329 0.115620 0.202445 13 0
1.6000000000E-01 1.0000000000E-01
0.034080 0.998234 0.118819 0.886971 16 3
2.2000000000E-01 5.8000000000E-01
0.056248 0.646713 0.112502 0.274274 22 1
3.4000000000E-01 2.6000000000E-01
0.073427 0.348604 0.190487 0.004924 45 2
8.0000000000E-01 4.2000000000E-01
0.089720 0.999701 0.233808 0.959723 12 5
1.4000000000E-01 9.0000000000E-01
0.061539 0.944619 0.143413 0.450240 34 4
5.8000000000E-01 7.4000000000E-01
0.036066 0.980446 0.129609 0.471078 29 3
4.8000000000E-01 5.8000000000E-01
0.094592 0.998731 0.237850 0.935265 10 4
1.0000000000E-01 7.4000000000E-01
0.054331 0.807188 0.123620 0.220503 42 4
7.4000000000E-01 7.4000000000E-01
0.095846 0.970252 0.203684 0.735934 13 3
1.6000000000E-01 5.8000000000E-01
0.035195 0.599010 0.172688 0.005624 39 1
6.8000000000E-01 2.6000000000E-01
0.086932 0.020028 0.106787 0.007781 43 0
7.6000000000E-01 1.0000000000E-01
0.012025 0.814110 0.198177 0.023406 17 0
2.4000000000E-01 1.0000000000E-01
0.040132 0.999027 0.243348 0.442101 20 4

Table 4A contd.

```
3.0000000000E-01 7.4000000000E-01
0.067393 0.981141 0.150388 0.742272   23   4
3.0000000000E-01 7.4000000000E-01
0.029067 0.975989 0.195423 0.044522   38   3
6.6000000000E-01 5.8000000000E-01
0.033447 0.945013 0.150933 0.093728   43   4
7.6000000000E-01 7.4000000000E-01
0.067777 0.980141 0.245585 0.421035   16   3
2.2000000000E-01 5.8000000000E-01
0.071050 0.997914 0.127680 0.965803   20   5
3.0000000000E-01 9.0000000000E-01
0.091794 0.180765 0.106069 0.121510   33   1
5.6000000000E-01 2.6000000000E-01
0.088011 0.488186 0.248671 0.003202   42   3
7.4000000000E-01 5.8000000000E-01
0.072163 0.122560 0.159412 0.002075   49   1
8.8000000000E-01 2.6000000000E-01
0.045607 0.974505 0.182439 0.169596   37   4
6.4000000000E-01 7.4000000000E-01
0.082242 0.933743 0.133168 0.713380   27   4
4.4000000000E-01 7.4000000000E-01
0.076854 0.278178 0.176997 0.044302   16   0
2.2000000000E-01 1.0000000000E-01
0.048847 0.744533 0.180812 0.100282   20   1
3.0000000000E-01 2.6000000000E-01
0.049227 0.108489 0.109667 0.006030   44   0
7.8000000000E-01 1.0000000000E-01
0.096961 0.577818 0.142726 0.265166   34   3
5.8000000000E-01 5.8000000000E-01
0.065131 0.496224 0.240431 0.003633   43   3
7.6000000000E-01 5.8000000000E-01
0.069559 0.998696 0.110280 0.990209   13   4
1.6000000000E-01 7.4000000000E-01
0.058389 0.985026 0.195817 0.566432   17   3
2.4000000000E-01 5.8000000000E-01
0.025181 0.742459 0.198215 0.001929   39   1
6.8000000000E-01 2.6000000000E-01
0.038421 0.439221 0.249107 0.002439   21   0
3.2000000000E-01 1.0000000000E-01
0.047987 0.104130 0.178293 0.000119   46   0
8.2000000000E-01 1.0000000000E-01
0.057175 0.581713 0.154307 0.037100   41   2
7.2000000000E-01 4.2000000000E-01
0.050062 0.973963 0.166328 0.311283   34   4
5.8000000000E-01 7.4000000000E-01
0.087908 0.647864 0.102389 0.572397   14   1
1.8000000000E-01 2.6000000000E-01
0.071433 0.822474 0.234878 0.045851   31   3
5.2000000000E-01 5.8000000000E-01
0.030519 0.996922 0.162224 0.361236   33   4
5.6000000000E-01 7.4000000000E-01
0.060493 0.325235 0.153545 0.049759   18   0
2.6000000000E-01 1.0000000000E-01
0.024527 0.972025 0.221395 0.003700   47   5
8.4000000000E-01 9.0000000000E-01
0.097132 0.096601 0.248320 0.000203   39   1
6.8000000000E-01 2.6000000000E-01
0.099566 0.653080 0.142013 0.333273   39   5
6.8000000000E-01 9.0000000000E-01
0.053199 0.744366 0.136929 0.151405   33   2
```

Table 4A contd...

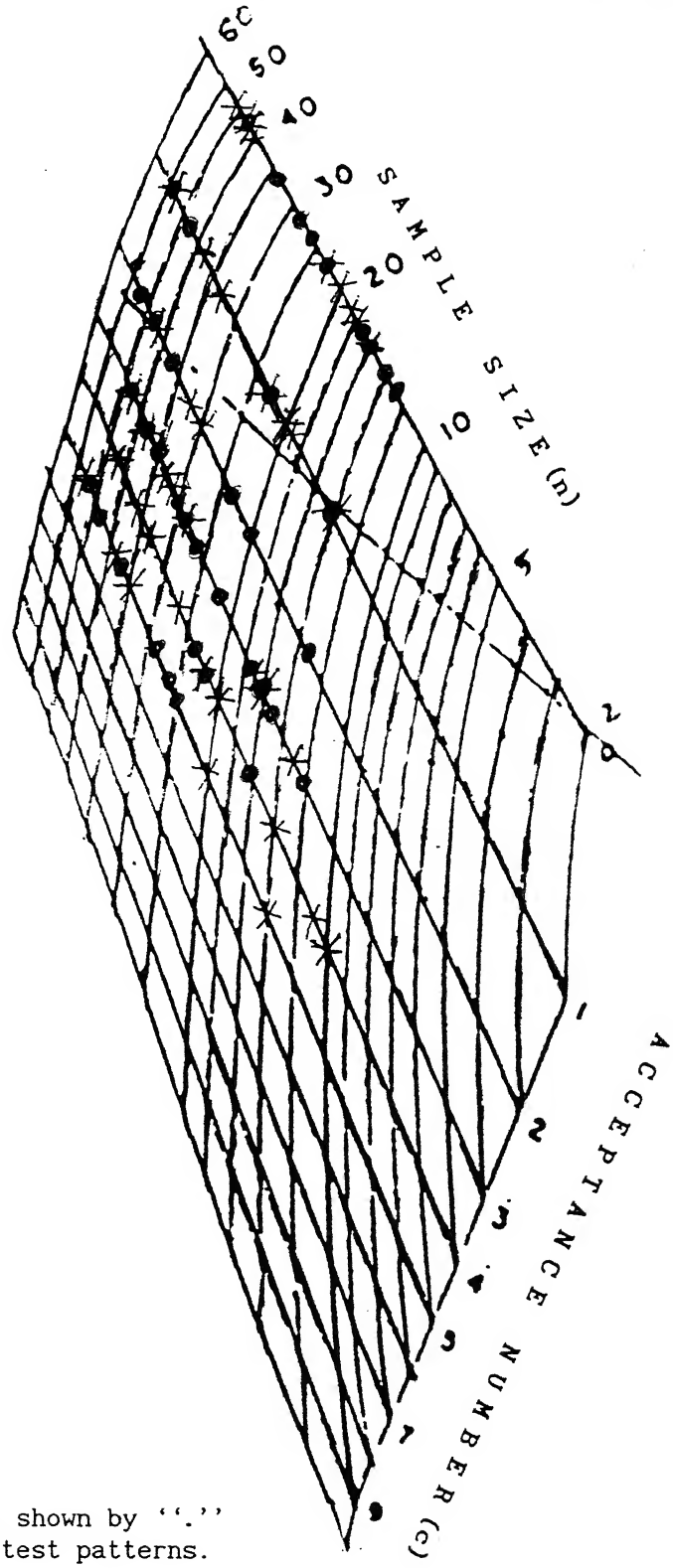
```
5.600000000000E-01 4.200000000000E-01
0.065246 0.998564 0.144147 0.959809    14     4
1.800000000000E-01 7.400000000000E-01
0.043601 0.734762 0.114886 0.240687    23     1
3.600000000000E-01 2.600000000000E-01
0.097757 0.383251 0.138479 0.135788    43     4
7.600000000000E-01 7.400000000000E-01
0.051629 0.999960 0.116872 0.996686    14     5
1.800000000000E-01 9.000000000000E-01
0.089696 0.115156 0.129302 0.041395    23     0
3.600000000000E-01 1.000000000000E-01
0.061007 0.675700 0.240875 0.037397    19     1
2.800000000000E-01 2.600000000000E-01
0.076956 0.871840 0.166760 0.283826    35     5
6.000000000000E-01 9.000000000000E-01
0.014242 0.555359 0.166572 0.000570    41     0
7.200000000000E-01 1.000000000000E-01
0.091990 0.998108 0.110665 0.995689    11     4
1.200000000000E-01 7.400000000000E-01
0.082291 0.546699 0.157086 0.128804    30     2
5.000000000000E-01 4.200000000000E-01
```

Table 5A: Prediction Set 3.

```
0.061539 0.944619 0.143413 0.450240   34    4
5.8000000000E-01 7.4000000000E-01
0.036066 0.980446 0.129609 0.471078   29    3
4.8000000000E-01 5.8000000000E-01
0.094592 0.998731 0.237850 0.935265   10    4
1.0000000000E-01 7.4000000000E-01
0.054331 0.807188 0.123620 0.220503   42    4
7.4000000000E-01 7.4000000000E-01
0.095846 0.970252 0.203684 0.735934   13    3
1.6000000000E-01 5.8000000000E-01
0.035195 0.599010 0.172688 0.005624   39    1
6.8000000000E-01 2.6000000000E-01
0.086932 0.020028 0.106787 0.007781   43    0
7.6000000000E-01 1.0000000000E-01
0.012025 0.814110 0.198177 0.023406   17    0
2.4000000000E-01 1.0000000000E-01
0.040132 0.999027 0.243348 0.442101   20    4
3.0000000000E-01 7.4000000000E-01
0.069393 0.981141 0.150388 0.742272   23    4
3.6000000000E-01 7.4000000000E-01
0.029067 0.975989 0.195423 0.044522   38    3
5.6000000000E-01 5.8000000000E-01
0.033447 0.945013 0.150933 0.093728   43    4
7.6000000000E-01 7.4000000000E-01
0.067777 0.980141 0.245585 0.421035   16    3
2.2000000000E-01 5.8000000000E-01
0.071050 0.997914 0.127680 0.965803   20    5
3.0000000000E-01 9.0000000000E-01
0.091794 0.180765 0.106069 0.121510   33    1
5.6000000000E-01 2.6000000000E-01
0.088011 0.488186 0.248671 0.003202   42    3
7.4000000000E-01 5.8000000000E-01
0.072163 0.122560 0.159412 0.002075   49    1
8.8000000000E-01 2.6000000000E-01
0.045607 0.974505 0.182439 0.169596   37    4
6.4000000000E-01 7.4000000000E-01
0.082242 0.933743 0.133168 0.713380   27    4
4.4000000000E-01 7.4000000000E-01
0.076854 0.278178 0.176997 0.044302   16    0
2.2000000000E-01 1.0000000000E-01
0.048847 0.744533 0.180812 0.100282   20    1
3.0000000000E-01 2.6000000000E-01
0.049227 0.108489 0.109667 0.006030   44    0
7.8000000000E-01 1.0000000000E-01
0.096961 0.577818 0.142726 0.265166   34    3
5.8000000000E-01 5.8000000000E-01
0.085131 0.496224 0.240431 0.003633   43    3
7.6000000000E-01 5.8000000000E-01
0.069559 0.998696 0.110280 0.990209   13    4
1.6000000000E-01 7.4000000000E-01
0.058389 0.985026 0.195817 0.566432   17    3
2.4000000000E-01 5.8000000000E-01
0.025181 0.742459 0.198215 0.001929   39    1
6.8000000000E-01 2.6000000000E-01
0.038421 0.439221 0.249107 0.002439   21    0
3.2000000000E-01 1.0000000000E-01
0.047987 0.104130 0.178293 0.000119   46    0
8.2000000000E-01 1.0000000000E-01
```

0.057175 0.581713 0.154307 0.037100 41 2
7.2000000000E-01 4.2000000000E-01
0.050062 0.973963 0.166328 0.311283 34 4
5.8000000000E-01 7.4000000000E-01
0.087908 0.647864 0.102389 0.572397 14 1
1.8000000000E-01 2.6000000000E-01
0.071433 0.822474 0.234878 0.045851 31 3
5.2000000000E-01 5.8000000000E-01
0.030519 0.996922 0.162224 0.361236 33 4
5.6000000000E-01 7.4000000000E-01
0.060493 0.325235 0.153545 0.049759 18 0
2.6000000000E-01 1.0000000000E-01
0.024527 0.972025 0.221395 0.003700 47 5
8.4000000000E-01 9.0000000000E-01
0.097132 0.096601 0.248320 0.000203 39 1
6.8000000000E-01 2.6000000000E-01
0.099566 0.653080 0.142013 0.333273 39 5
6.8000000000E-01 9.0000000000E-01
0.053199 0.744366 0.136929 0.151405 33 2
5.6000000000E-01 4.2000000000E-01
0.065246 0.998564 0.144147 0.959809 14 4
1.8000000000E-01 7.4000000000E-01
0.043601 0.734762 0.114886 0.240687 23 1
3.6000000000E-01 2.6000000000E-01
0.097757 0.383251 0.138479 0.135788 43 4
7.6000000000E-01 7.4000000000E-01
0.051629 0.999960 0.116872 0.996686 14 5
1.8000000000E-01 9.0000000000E-01
0.089696 0.115156 0.129302 0.041395 23 0
3.6000000000E-01 1.0000000000E-01
0.061007 0.675700 0.240875 0.037397 19 1
2.8000000000E-01 2.6000000000E-01
0.076956 0.871840 0.166760 0.283826 35 5
6.0000000000E-01 9.0000000000E-01
0.014242 0.555359 0.166572 0.000570 41 0
7.2000000000E-01 1.0000000000E-01
0.091990 0.998108 0.110665 0.995689 11 4
1.2000000000E-01 7.4000000000E-01
0.082291 0.546699 0.157086 0.128804 30 2
5.0000000000E-01 4.2000000000E-01
0.048293 0.856471 0.212708 0.013096 42 3
7.4000000000E-01 5.8000000000E-01
0.051234 0.109822 0.199863 0.000086 42 0
7.4000000000E-01 1.0000000000E-01

Fig. 3A: Nomograph with patterns in Table 4A and 5A.

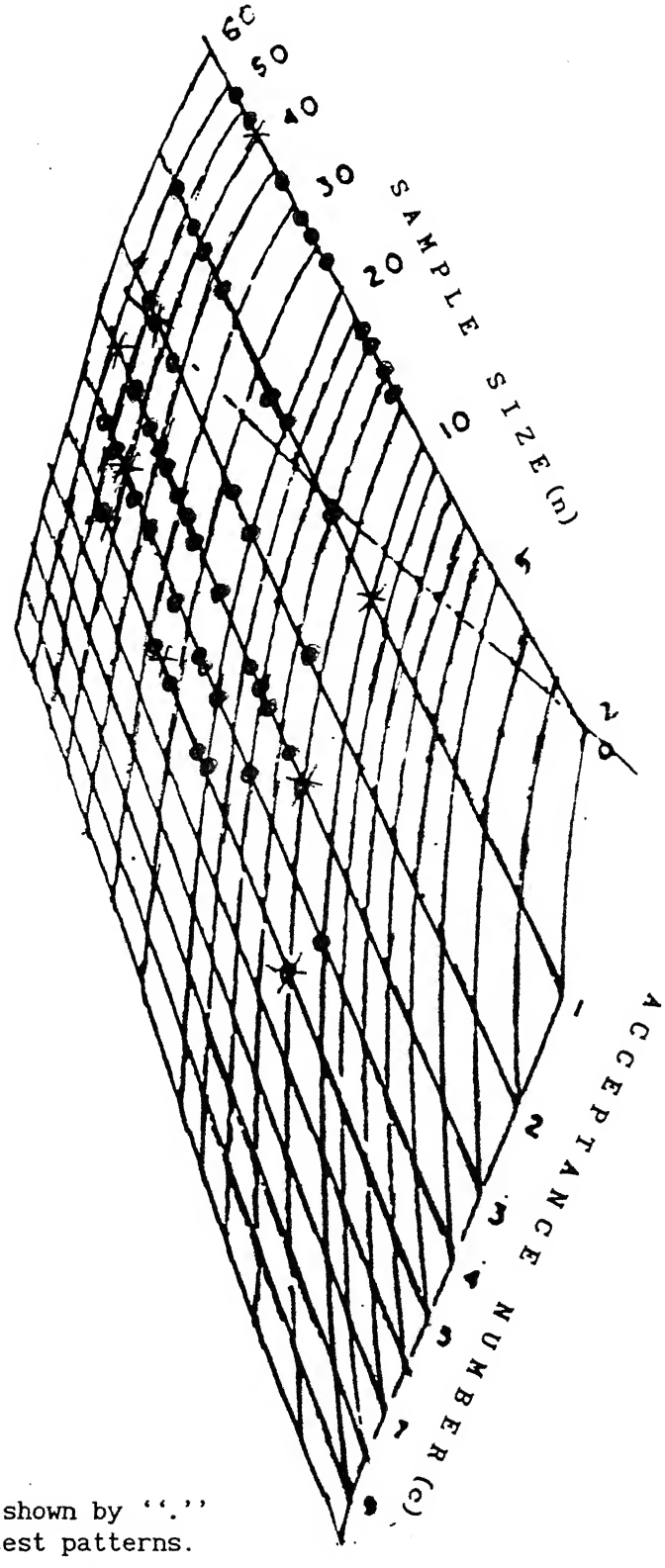


The training patterns are shown by ''.'' and, ''**x**'' represent the test patterns.

Table 6A: Prediction Set 4.

```
0.027483 0.974242 0.207356 0.018836   41    4
7.2000000000E-01 7.4000000000E-01
0.065802 0.625840 0.163872 0.038626   47    3
8.4000000000E-01 5.8000000000E-01
0.037080 0.999608 0.236097 0.359371   27    5
4.4000000000E-01 9.0000000000E-01
0.057005 0.552972 0.199905 0.004659   43    2
7.6000000000E-01 4.2000000000E-01
0.026645 1.000000 0.186032 0.986333   12    5
1.4000000000E-01 9.0000000000E-01
0.072142 0.046425 0.216744 0.000045   41    0
7.2000000000E-01 1.0000000000E-01
0.056319 0.894342 0.224511 0.306390   10    1
1.0000000000E-01 2.6000000000E-01
0.051011 0.824882 0.217669 0.009083   43    5
7.6000000000E-01 9.0000000000E-01
0.021070 0.999942 0.242610 0.733440   11    3
1.2000000000E-01 5.8000000000E-01
0.051588 0.946363 0.221351 0.040507   40    4
7.0000000000E-01 7.4000000000E-01
```

Fig. 4A: Nomograph with patterns in Table 4A and 8A.



The training patterns are shown by ''.'' and, ''**x**'' represent the test patterns.

APPENDIX B

```
Program ANN(input,output);
{
   An with 2 hidden layer is taken whose input
 layer is referred to by alphabet a, the first
 hidden layer by b, the second  hidden layer
 by c, and the output layer by d
}
const
  nmax1 = 200;
type
  matrix2 = ARRAY[1..150,1..150] of real;

var
    a,b,c,d,pattern_no,seed,iter: INTEGER;
    last_pat,counter, dchoice,oscale_1,oscale_2 : INTEGER;
    n_in,n_out,n_hid1,n_hid2,n_layers, n0, n1, n2 : INTEGER;
    y1,y2 : ARRAY [1..150] of real;
    delta,error_sum : ARRAY [1..nmax1] of real;
    target,y3 : ARRAY [1..nmax1,1..2] of real;
    pre_del_w3,pre_del_w2,w3,w2 : matrix2;
    pre_del_w1,w1 : matrix2;
    in_data,del_w : ARRAY [1..nmax1,1..150] of real;
    cutoff_n,cutoff_c,iscale,rmse,actual_n,actual_c,
    beta,eta,alpha,x1,pssd,sse,accept_no,sample_size : real;
    glix1,glix2,glix3 : longint;
    glr : ARRAY[1..97] of real;
    pfile, ifile,pfile1 : STRING[25];
    trained,fine,bias_node : boolean;
    infile,outfile,infile1,results,answers : TEXT;
    {-----------------------------------------------------}
FUNCTION ran(VAR idum:integer):real;
 CONST
   m1 = 259200;
   ia1 = 7141;
   ic1 = 54773;
   rm1 = 1/m1;
   m2 = 134456;
   ia2 = 8121;
   ic2 = 28411;
   rm2 = 1/m2;
   m3 = 243000;
   ia3 = 4561;
   ic3 = 51349;
   {m1,m2,m3,ia1,ia2,ia3,ic1,ic3,ic3 : longint; }
 VAR
   j : integer;
 BEGIN
   IF (idum < 0) THEN
   BEGIN
      glix1 := (ic1 - idum) MOD m1;
      glix1 := (ia1 * glix1 + ic1) MOD m1;
      glix2 := glix1 MOD m2;
      glix1 := (ia1 * glix1 + ic1) MOD m1;
      glix3 := glix1 MOD m3;
      FOR j := 1 TO 97 DO
      BEGIN
         glix1 := (ia1 * glix1 + ic1) MOD m1;
         glix2 := (ia2 * glix2 + ic2) MOD m2;
```

```
            glr[j] := (glix1 + glix2 * rm2) * rm1;
          END;
          idum := 1;
      END;
      glix1 := (ia1 * glix1 + ic1) MOD m1;
      glix2 := (ia2 * glix2 + ic2) MOD m2;
      glix3 := (ia3 * glix3 + ic3) MOD m3;
      j := 1 + (97 * glix3) DIV m3;
      IF ( j > 97 ) OR (j < 1 ) THEN
      BEGIN
       writeln('Error ** from random number generator');
      END;
      ran := glr[j];
      glr[j] := (glix1 + glix2 * rm2) * rm1;
END;{of ran}
{-------------------------------------------------------}
PROCEDURE mread(VAR ra:matrix2;arow,acol:integer);
{assigns weights to the arcs from the "WEIGHTS" file}
 var
  row,col:integer;
 begin
  for row:=1 to arow do
   begin
    for col:=1 to acol do
     begin
      read(infile,ra[row,col])
     end;
    readln(infile);
   end
 end;{of mread}
{-------------------------------------------------------}
PROCEDURE describe_the_topology;
{
  This procedure reads the structure of the ANN,
the training parameters
}
 var
 a,b,c,d,pattern_no :integer;
  ch : char;
 begin
{
  Read structure of ANN
}
  n_hid1 := 0;n_hid2 := 0; n_layers := 0;
  n_in := 0; n_out := 0;
  writeln('*** DESCRIBE STRUCTURE OF ANN ***'); writeln;
  write('1.  No. nodes in INPUT layer : ');
  readln(n_in);
  write('2.  No. nodes in OUTPUT layer : ');
  readln(n_out);
  write('3. Input beta factor of sigmoid function : ');
  readln(beta);
  write('4.  No. hidden layers : ');
  readln(n_layers);
  if (n_layers > 0) then
   begin
    write('5.  No. nodes in FIRST HIDDEN layer : ');
    readln(n_hid1);
```

```
      if (n_layers > 1) then
       begin
        write('6.  No. nodes in SECOND HIDDEN layer : ');
        readln(n_hid2);
       end;
     end;
    if (n_hid1 = 0) then
     n_hid1 := n_out;
    if(n_hid2 = 0) then
     n_hid2 := n_out;
    ch := 'y';
    write('7. Do you wish to include bias node (y or n) : ');
    readln(ch);
    if ((ch = 'Y') or (ch = 'y')) then
     bias_node := true
    else
     bias_node := false;
     if (bias_node) then
      begin
       n0 := n_in + 1;
       n1 := n_hid1 + 1;
       n2 := n_hid2 + 1;
{
Set the output of bias node to be equal to 1
by treating it as an additional node in each
layer except output layer
}
       y1[n1] := 1.0;
       y2[n2] := 1.0;
      end
     else
      begin
       n0 := n_in;
       n1 := n_hid1;
       n2 := n_hid2;
      end;
{
  Read the training parameters
}
  write('Do you wish to train (= 0), resume training (= 1)
         or test (= 2)? ');
  readln(dchoice);
  if(dchoice < 2) then
   begin
    writeln('*** INPUT THE TRAINING PARAMETERS ***');
    writeln;
    write('value of iteration to proceed with: ');
    readln(iter);
    write('What is the maximum permissible iterations : ');
    readln(counter);
    write('Input the learning rate : ');
    readln(eta);
    write('Input the momentum factor : ');
    readln(alpha);
    write('Input file containing training patterns : ');
    readln(pfile);
    write('Input the number of training patterns : ');
    readln(last_pat);
```

```
      write('Input the error tolerance for trai
      readln(cutoff_n);
      write('Input the error tolerance for trai
      readln(cutoff_c);
      write('scaling factor for scaling inputs
      readln(iscale);
     end;
     if(dchoice <> 0) then
      begin
       reset(infile,'WEIGHTS');
       mread(w1,n0,n_hid1);
       If(n_layers > 0) then
        mread(w2,n1,n_hid2);
       if(n_layers > 1) then
        mread(w3,n2,n_out);
       close(infile);
{
       writeln('WEIGHTS');
       for a := 1 to n0 do
        write(w1[a,1]:5:2); writeln;
       for b:= 1 to n1 do
        write(w2[b,1]:5:2); writeln;
       for c := 1 to n2 do
       write(w3[c,1]:5:2); writeln;
}
     end; {If dchoice<>0}
    for a:=1 to n0 do
     for b:=1 to n_hid1 do
      pre_del_w1[a,b]:=0;
    for b:=1 to n1 do
     for c:=1 to n_hid2 do
      pre_del_w2[b,c]:=0;
    for c:=1 to n2 do
     for d:=1 to n_out do
      pre_del_w3[c,d]:=0;
 end;{of describe_the_topology}
{-----------------------------------------------
PROCEDURE initialize_the_ANN;
{
  This procedure initializes the weights
using random numbers between [-0.5,0.5]
}
var a, b,c,d,pattern_no :integer;
 begin
  seed := - 1;
  for a:=1 to n0 do
  begin
   for b:=1 to n_hid1 do
    w1[a,b] := 0.5*ran(seed);
  end;
 if(n_layers > 0) then
  begin
   for b:=1 to n1 do
    begin
     for c:=1 to n_hid2 do
      w2[b,c] := 0.5*ran(seed);
    end;
  end;
```

```
  if(n_layers > 1) then
   begin
    for c:=1 to n2 do
     begin
      for d:=1 to n_out do
       w3[c,d] := 0.5*ran(seed);
     end;
   end;
end; {initialize_the_ANN}
{-------------------------------------------------------------}
PROCEDURE read_data_to_train;
{
  This procedure reads the input data
  and target values for all training patterns
}
var a,b,c,d,pattern_no :integer;
 begin
  for pattern_no:=1 to last_pat do
   begin
{
  Read input data for pattern p
}
    for a:=1 to n_in do
     read(infile,in_data[pattern_no,a]);
    readln(infile);
{
  Read target values for pattern p
}
    for d:=1 to n_out do
     read(infile,target[pattern_no,d]);
    readln(infile);

   { writeln('Input patterns');
    for a:=1 to n_in do
     write(in_data[pattern_no,a]);
    writeln;
    writeln('target values');
    for d := 1 to n_out do
     write(target[pattern_no,d]);
    writeln;}

    for a:=1 to n_in do
    in_data[pattern_no,a]:=iscale*in_data[pattern_no,a];
    if (bias_node) then
     in_data[pattern_no,n0] := 1.0;
  end;
 end;{of read_data_to_train}
{-------------------------------------------------------------}
PROCEDURE forwardpass(p:integer);
{
  This procedure computes the output values
for the p'th pattern of the input data. The
output values are stored in  p'th row vector of y3
}
var a,b,c,d,pattern_no :integer;
  begin
{
  Compute outputs from first hidden layer
```

```
}
 for b:=1 to n_hid1 do
  begin
   x1 := 0.0;
   for a:=1 to n_in do
    x1:=x1+in_data[p,a]*w1[a,b];
   if (bias_node) then
    x1 := x1 + w1[n_in+1,b];
   x1 := x1*beta;
   if(abs(x1) > 25) then
    if (x1 < 0) then
     y1[b] := 0.0
    else
     y1[b] := 1.0
   else
    y1[b]:=1/(1+exp(-x1));
  end;
{
  Compute outputs of second hidden layer
}
 if (n_layers > 0) then
  begin
   for c:=1 to n_hid2 do
    begin
     x1 := 0.0;
     for b:=1 to n_hid1 do
      x1:=x1+y1[b]*w2[b,c];
     if (bias_node) then
      x1 := x1 + w2[n_hid1+1,c];
     x1 := x1*beta;
     if(abs(x1) > 25) then
      if (x1 < 0) then
       y2[c] := 0.0
      else
       y2[c] := 1.0
     else
      y2[c]:=1/(1+exp(-x1));
    end;
  end;
{
  Compute outputs from output layer
}
 if(n_layers > 1) then
  begin
   for d:=1 to n_out do
    begin
     x1 := 0.0;
     for c:=1 to n_hid2 do
      x1:=x1+y2[c]*w3[c,d];
     if (bias_node) then
      x1 := x1 + w3[n_hid2+1,d];
     x1 := x1*beta;
     if(abs(x1) > 25) then
      if (x1 < 0) then
       y3[p,d] := 0.0
      else
       y3[p,d] := 1.0
     else
```

```
        y3[p,d]:=1/(1+exp(-x1));
      end;
    end;

    CASE n_layers of
      0 : for d := 1 to n_out do y3[p,d] := y1[d];
      1 : for d := 1 to n_out do y3[p,d] := y2[d];
    END;
  end;{of forwardpass}
{-------------------------------------------------------------}
PROCEDURE Backprop(p:integer);
{
    This procedure applies the back-propogation algorithm
 for adjusting the weights, given the target values and
output values predicted by the ANN for a pattern.  The
output values are assumed to be stored in p'th row
vector of y3 and the target values in p'th row vector
of target.
}
var a,b,c,d,pattern_no :integer;
 begin
  for d:=1 to n_out do
  delta[d]:= beta*y3[p,d]*(1-y3[p,d])*(target[p,d]-y3[p,d]);
{
  Change the weights on connections between
second hidden layer and output layer
}
  if(n_layers > 1) then
   begin
    for c:=1 to n2 do
     begin
      error_sum[c] := 0.0;
      for d:=1 to n_out do
      begin
       del_w[c,d]:=delta[d]*y2[c]*eta+alpha*pre_del_w3[c,d];
       error_sum[c] := error_sum[c]+w3[c,d]*delta[d];
       w3[c,d]:=w3[c,d]+del_w[c,d];
       pre_del_w3[c,d]:=del_w[c,d]
      end;
     end;
    for c := 1 to n_hid2 do
     delta[c]:= beta*y2[c]*(1-y2[c])* error_sum[c];
   end;
{
Change the weights on connections between
 fisrt and second hidden layer
}
   if(n_layers > 0) then
    begin
     for b:=1 to n1 do
      begin
       error_sum[b] := 0.0;
       for c:=1 to n_hid2 do
        begin
        del_w[b,c]:=delta[c]*y1[b]*eta+alpha*pre_del_w2[b,c];
        error_sum[b] := error_sum[b]+w2[b,c]*delta[c];
        w2[b,c]:=w2[b,c]+del_w[b,c];
        pre_del_w2[b,c]:=del_w[b,c]
```

```
          end;
        end;
       for b := 1 to n_hid1 do
       delta[b]:= beta*y1[b]*(1-y1[b])* error_sum[b];
      end;
{
  Change the weights of connections between
 input layer and first hidden layer
}
for a:=1 to n0 do
 begin
  for b:=1 to n_hid1 do
   begin
   del_w[a,b]:=delta[b]*in_data[p,a]*eta +
                              alpha*pre_del_w1[a,b];
   w1[a,b]:=w1[a,b]+del_w[a,b];
   pre_del_w1[a,b]:=del_w[a,b]
   end
  end;
end;{of backprop}
{-------------------------------------------------------------}
FUNCTION test_of_closeness : boolean;
{
  Checks for convergence on each output node
in each  iteration for each pattern.
}
var a,b,c,d,pattern_no :integer;
 begin
  test_of_closeness := true;
  for pattern_no:=1 to last_pat do
  if
(((abs(target[pattern_no,1]-y3[pattern_no,1]))>cutoff_n) or
((abs(target[pattern_no,2]-y3[pattern_no,2])) > cutoff_c))
  then
  test_of_closeness:=false;
  if ((iter = 1) or(iter mod 1000 =0)) then
    begin
     pssd:=0.0;
     for pattern_no:=1 to last_pat do
     begin
      for d:=1 to n_out do
       begin
        sse:= sqr(target[pattern_no,d]-y3[pattern_no,d]);
        pssd:= pssd + sse;
       end;
     end;
      writeln(iter:6,pssd);
     end;
 end;{of test_of_closeness}
{-------------------------------------------------------------}
 PROCEDURE do_predictions;
{
     does predictions by using the WEIGHTS file so obtained.
}
 var a,b,c,d,pattern_no,new_last:integer;
     snp,sna,ersn,scp,sca,ersc,
     np,na,ern,cp,ca,erc,pern,perc,persn,persc:real;
```

```
  begin
   writeln('In a predict procedure');
   writeln('File name containing input pattern');
   readln(pfile1);
   write('Input scaling factor to scale output1: ');
   readln(oscale_1);
   writeln;
   write('Input scaling factor to scale output2: ');
   readln(oscale_2);
   writeln;
   writeln('Number of patterns ');
   readln(new_last);
   writeln;
   writeln(' PREDICTING FOR FILE NOT USED IN TRAINING ');
   reset(infile1,pfile1);
   for pattern_no:=1 to new_last do
   begin
     y3[pattern_no,1]:=0.0;
     y3[pattern_no,2]:=0.0;
   end;
   for pattern_no:=1 to new_last do
    begin {reading patterns}
     for a:=1 to n_in do
       read(infile1,in_data[pattern_no,a]);
     readln(infile1);
     for d:=1 to n_out do
       read(infile1,target[pattern_no,d]);
     readln(infile1);
    end;
    writeln;
  writeln('     ********   SCALED   ******** ');
writeln('-----------------------------------------');
    writeln('targ_n  pred_n  err_n  perr_n  targ_c  pred_c
err_c  perr_c ');
writeln('-----------------------------------------');
  for pattern_no:=1 to new_last do
  begin
    forwardpass(pattern_no);
    snp:=y3[pattern_no,1];
    sna:=target[pattern_no,1];
    ersn:=(snp-sna);
    persn:=abs(ersn/sna)*100;
    scp:=y3[pattern_no,2];
    sca:=target[pattern_no,2];
    ersc:=(scp-sca);
    persc:=abs(ersc/sca)*100;
    writeln(sna:6:4,'  ',snp:6:4,'  ',ersn:7:4,'  ',persn:6:2,'
',sca:6:4,'  ',scp:6:4,'  ',ersc:7:4,'  ',persc:6:2);
    end;
writeln('-----------------------------------------');
    pssd:=0.0;
    for pattern_no:=1 to new_last do
    begin
     for d:=1 to n_out do
      begin
        sse:= sqr(target[pattern_no,d]-y3[pattern_no,d]);
        pssd:= pssd + sse;
      end;
```

```
    end;
    writeln('          pssd = ',pssd);
    writeln;
  writeln(' ***********   UNSCALED   **********');
writeln('-----------------------------------------');
    writeln('targ_n  pred_n  err_n  perr_n  targ_c  pred_c
err_c  perr_c ');
writeln('-----------------------------------------');
  for pattern_no:=1 to new_last do
  begin
    y3[pattern_no,1]:=0.0;
    y3[pattern_no,2]:=0.0;
  end;
  for pattern_no:=1 to new_last do
  begin
   forwardpass(pattern_no);
   np:=(y3[pattern_no,1]*oscale_1);
   na:=(target[pattern_no,1]*oscale_1);
   ern:=(np-na);
   pern:=abs(ern/na)*100;
   cp:= (y3[pattern_no,2]*oscale_2);
   ca:=(target[pattern_no,2]*oscale_2);
   erc:=(cp-ca);
   perc:=abs(erc/ca)*100;
   writeln(na:6:1,' ',np:6:1,' ',ern:7:1,' ',pern:6:2,
' ',ca:6:1,' ',cp:6:1,' ',erc:7:1,' ',perc:6:2);
  end;
writeln('-----------------------------------------');
end;{of do_predictions}
{-------------------------------------------------------}
PROCEDURE update_weights;
var a,b,c,d,pattern_no :integer;
 begin
  for a := 1 to n0 do
   begin
    for b:=1 to n_hid1 do
     begin
      write(outfile,w1[a,b]:6:3,' ');
     end;
    writeln(outfile);
   end;
  if(n_layers > 0) then
   for b:=1 to n1 do
    begin
     for c:=1 to n_hid2 do
      write(outfile,w2[b,c]:6:3,' ');
     writeln(outfile);
    end;
   if(n_layers > 1) then
    for c:=1 to n2 do
     begin
      for d:=1 to n_out  do
       write(outfile,w3[c,d]:6:3,' ');
      writeln(outfile);
     end;
 end;{of update_weights}
{-------------------------------------------------------}
PROCEDURE tell_iterations_done;
```

```
{
keeps on informing the user as to how  many  iterations
have been done in multiples of 5.
}
var a,b,c,d,pattern_no :integer;
 begin
  if (iter mod 5=0) then
   begin
    writeln;
    writeln('iteration no. = ',iter:3);
   end;
 end;{tell_iterations_done}
{-------------------------------------------------------------}
PROCEDURE list_output_of_ANN;
{
 sends the output on the screen/file during the training.
}
var a,b,c,d,pattern_no :integer;
 begin
    writeln;
    writeln('-------------------------------------');
    writeln('targ_n  pred_n  err_n  targ_c  pred_c  err_c ');
    writeln('-------------------------------------');
  for pattern_no:=1 to last_pat do
   begin
    write(target[pattern_no,1]:6:4,y3[pattern_no,1]:6:4);
    write((target[pattern_no,1]-y3[pattern_no,1]):7:4,'  ');
    write(target[pattern_no,2]:6:4,y3[pattern_no,2]:6:4);
    write((target[pattern_no,2]-y3[pattern_no,2]):7:4);
    writeln;
   end;
    writeln('-------------------------------------');
 end;{of list_output_of_ANN}
{-------------------------------------------------------------}
PROCEDURE prepare_ANN;
{
     Once the test of closeness condition gets
satisfied on  each node this procedure updates
the WEIGHTS file for subsequent use.
}
var a,b,c,d,pattern_no :integer;
begin
 writeln;
 writeln(output,'convergence ACHIEVED at iter no.',iter:4);
 pssd:=0.0;
  for pattern_no:=1 to last_pat do
    begin
     for d:=1 to n_out do
      begin
        sse:= sqr(target[pattern_no,d]-y3[pattern_no,d]);
        pssd:= pssd + sse;
      end;
    end;
   writeln('The pssd finally was ',pssd);
   writeln('-------------------------------------');
   writeln('targ_n  pred_n  err_n  targ_c  pred_c  err_c ');
   writeln('-------------------------------------');
   for pattern_no:=1 to last_pat do
```

```
    begin
     write(target[pattern_no,1]:6:4,y3[pattern_no,1]:6:4);
     write((target[pattern_no,1]-y3[pattern_no,1]):7:4);
     write(target[pattern_no,2]:6:4,y3[pattern_no,2]:6:4);
     write((target[pattern_no,2]-y3[pattern_no,2]):7:4);
     writeln;
    end;
     writeln('-------------------------------------');
 end;{of prepare_ANN}
{-------------------------------------------------------}
FUNCTION iter_is_multiple_of_five:boolean;
{
 serves as a counter.
}
 begin
  iter_is_multiple_of_five := false;
  if (iter mod 5 = 0) then
   iter_is_multiple_of_five := true
 end;{of iter_is_multiple_of_hundred}
{-------------------------------------------------------}
 BEGIN {THE MAIN PROGRAM}
  describe_the_topology;
  IF  (dchoice < 2) THEN
   BEGIN  {training}
     reset(infile,pfile);
     IF  (dchoice = 0) THEN initialize_the_ANN;
     read_data_to_train;
     trained:= false;
     WHILE  ((not trained) and (iter<=counter))  DO
      BEGIN
        FOR pattern_no := 1 to last_pat DO
        BEGIN
          forwardpass(pattern_no);
          backprop(pattern_no);
        END;
        iter := iter + 1;
        list_output_of_ANN;
        tell_iterations_done;
        IF iter_is_multiple_of_five THEN
        BEGIN
          rewrite(outfile,'WEIGHTS');
          update_weights;
          close(outfile);
        END;
        fine := test_of_closeness;
```

```
        IF fine THEN
        BEGIN
          prepare_ANN;
          trained:= true
        END;
      END; { of while loop}
    END {of training}
  ELSE do_predictions;
END.{ OF PROGRAM }
```

A regression analysis of the data contained in Tables 7.1, 7.2, 7.3 and 7.4 was done in order to get the functional relationships between the actual and predicted values of n (and c). These are reported here.

Table 7.1

Regression on n:

The equation of curve of best fit is

Pred-n = 0.875789(Actual-n) + 2.994966

The Standard error on Y Est = 5.971333, and

the R Squared value = 0.752113.

Regression on c:

The equation of curve of best fit is

Pred-c = 0.530303(Actual-c) + 1.373939

The Standard error on Y Est = 0.890284, and

the R Squared value = 0.422562.

Table 7.2

Regression on n:

The equation of curve of best fit is

Pred-n = 1.008709(Actual-n) + 0.201016

The Standard error on Y Est = 1.800255, and

the R Squared value = 0.984363.

Regression on c:

The equation of curve of best fit is

Pred-c = 0.991176(Actual-c) + 0.161765

The Standard error on Y Est = 0.269122, and

the R Squared value = 0.971902.

Table 7.3

Regression on n:

The equation of curve of best fit is

Pred-n = 1.055511(Actual-n) - 3.65459

The Standard error on Y Est = 3.090225, and

the R Squared value = 0.967487.

Regression on c:

The equation of curve of best fit is

Pred-c = 0.831594(Actual-c) - 0.1971

The Standard error on Y Est = 1.064101, and

the R Squared value = 0.678152.

Table 7.4

Regression on n:

The equation of curve of best fit is

Pred-n = 0.766037(Actual-n) + 2.826863

The Standard error on Y Est = 7.864872, and

the R Squared value = 0.567118.

Regression on c:

The equation of curve of best fit is

Pred-c = 0.935671(Actual-c) - 0.10229

The Standard error on Y Est = 1.167605, and

the R Squared value = 0.655908.

Conclusion:

In ideal case the equation of the fitting curve should be such that it passes through the origin of the coordinate axis. In other words, the y-intercept should be exactly zero and the standard error on Y must be as small as possible. Further, the R Squared value must be +1 or -1.

Thus, from the above results one can infer that the experiments on Table 7.2 ,approximates closely the ideal curve equation amongst the Tables analyzed here having R Squared value of 0.984363 and 0.971902 for n and c respectively. Noting the fact that the Prediction file tested upon, corresponding to this case consisted of patterns that merely matched the patterns taken to train the ANN, one can say that the number of training patterns has to be increased significantly (500 or more) to adequately cover the nomogram grid order to get a better predicting ANN finally.

Table 7.1: 12–node ANN trained on 50 patterns with lng rt 0.9 and mom. fac 0.1

Actual–n	Predctd–n	Error–n	% error in n	Actual–c	Predctd–c	Error–c	% error in c
24	27.7	3.7	15.31	2	2.3	0.3	13.4
7	6.8	−0.2	3.46	5	4.9	−0.1	2.81
44	43.2	−0.8	1.82	5	3.3	−1.7	33.7
23	17.1	−5.9	25.8	3	2.2	−0.8	26.69
44	40.8	−3.2	7.29	3	2.9	−0.1	3.18
26	15.8	−10.2	39.12	5	2.8	−2.2	43.27
22	30.7	8.7	39.41	4	4.9	0.9	23.49
17	21	4	23.48	3	3.9	0.9	28.4
24	20.4	−3.6	14.99	4	3.5	−0.5	12.56
23	28.9	5.9	25.53	1	1.6	0.6	59.11

Regression on n:

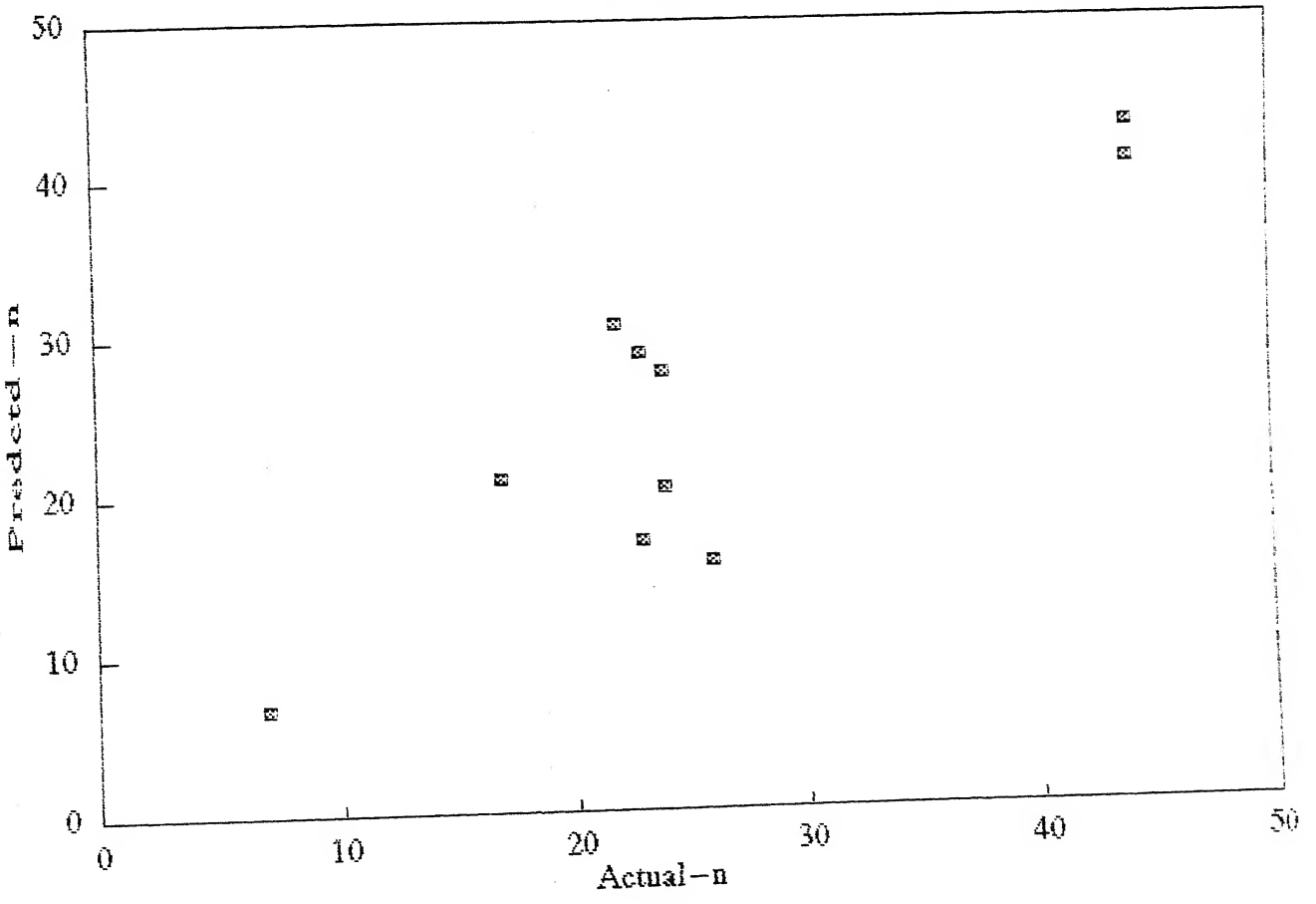
Constant	2.994966
Std Err of Y Est	5.971333
R Squared	0.752113
No. of Observations	10
Degrees of Freedom	8
X Coefficient(s)	0.875789
Std Err of Coef.	0.177762

Regression on c:

Constant	1.373939
Std Err of Y Est	0.890284
R Squared	0.422562
No. of Observations	10
Degrees of Freedom	8
X Coefficient(s)	0.530303
Std Err of Coef.	0.219173

SCATTER PLOT FOR SAMPLE SIZE (n)

in Table 7.1



SCATTER PLOT FOR c

in Table 7.1

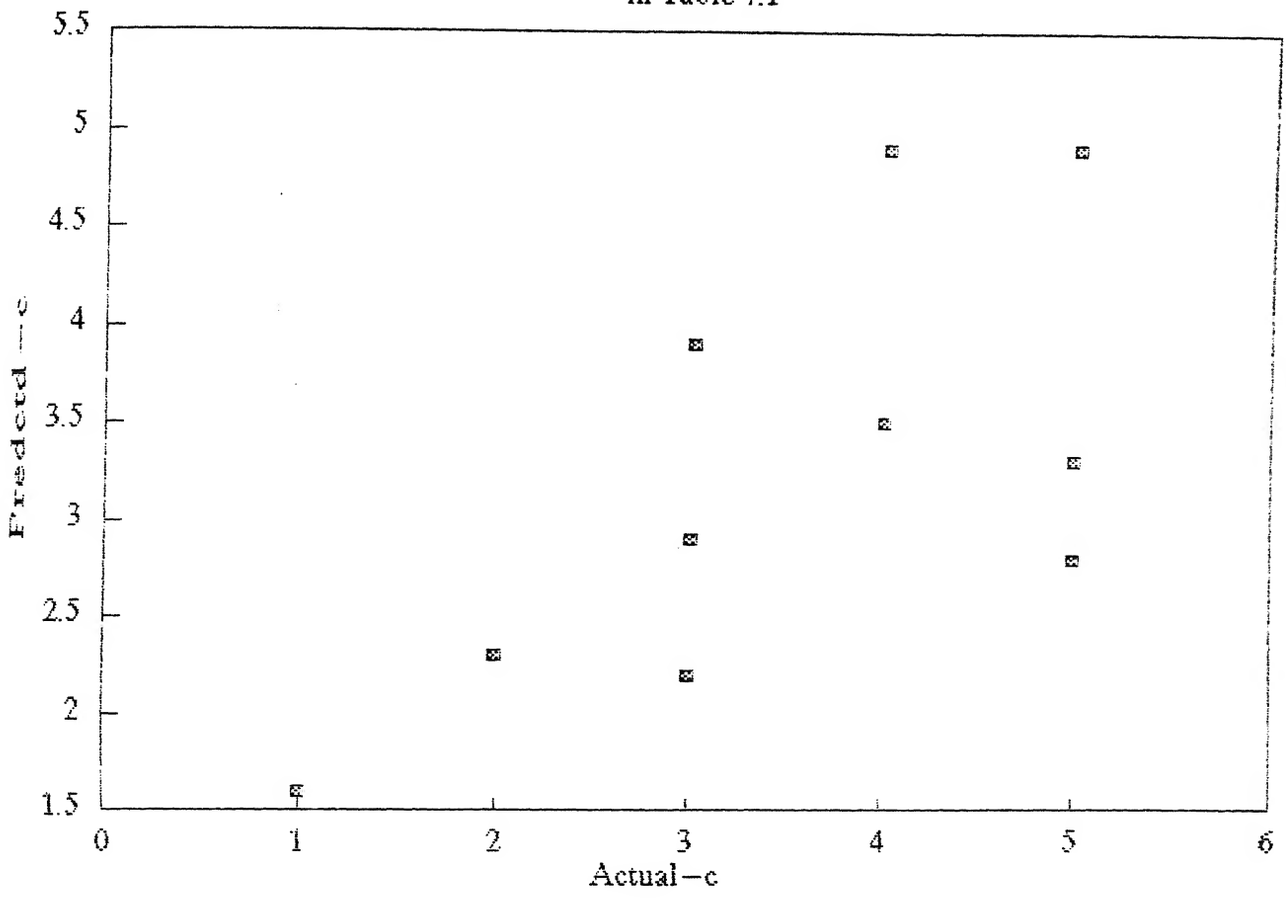


Table 7.2: 12–node ANN trained with 0.9 lng and 0.1 mom used to predict on P2.

Actual–n	Predctd–n	Error in n	% error in n	Actual–c	Predctd–c	Error in c	% error in c
12	9.4	−2.6	21.33	5	4.6	−0.4	8.95
34	33.5	−0.5	1.35	4	4.3	0.3	6.58
29	26.5	−2.5	8.49	3	2.9	−0.1	4.57
10	10.8	0.8	7.95	4	4.5	0.5	12.37
42	43.5	1.5	3.45	4	4.3	0.3	8.7
7	8.2	1.2	17.73	5	5.1	0.1	2.12
39	41.2	2.2	5.57	1	1	0	0.11
7	8.5	1.5	20.91	5	5.1	0.1	2.14
17	17.8	0.8	4.94	1	1.2	0.2	17.73
20	21.5	1.5	7.37	4	4.3	0.3	7.07

Regression on n:

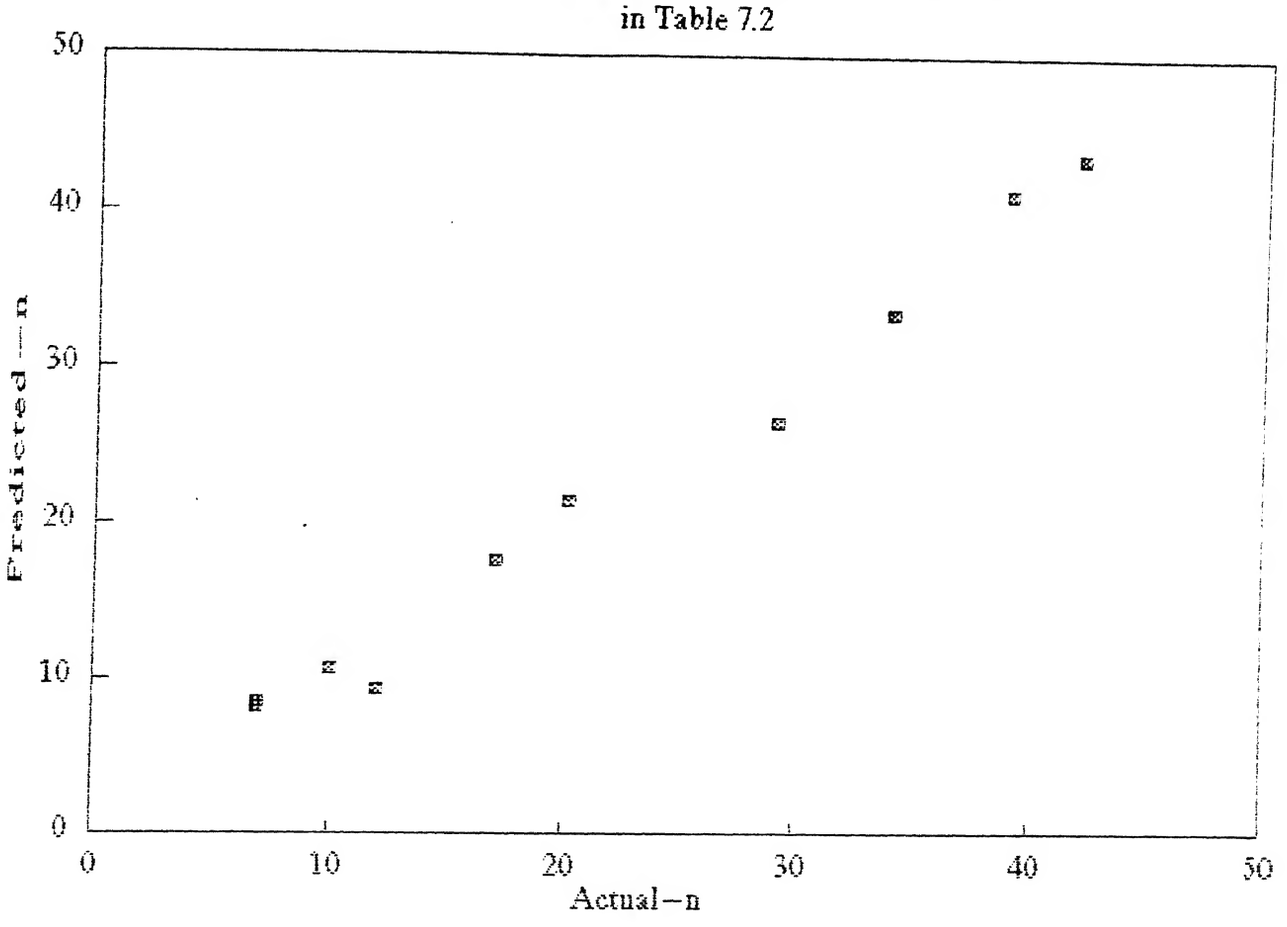
Constant	0.201016
Std Err of Y Est	1.800255
R Squared	0.984363
No. of Observations	10
Degrees of Freedom	8
X Coefficient(s)	1.008709
Std Err of Coef.	0.044949

Regression on c:

Constant	0.161765
Std Err of Y Est	0.269122
R Squared	0.971902
No. of Observations	10
Degrees of Freedom	8
X Coefficient(s)	0.991176
Std Err of Coef.	0.059585

SCATTER PLOT FOR n

in Table 7.2



SCATTER PLOT FOR c

in Table 7.2

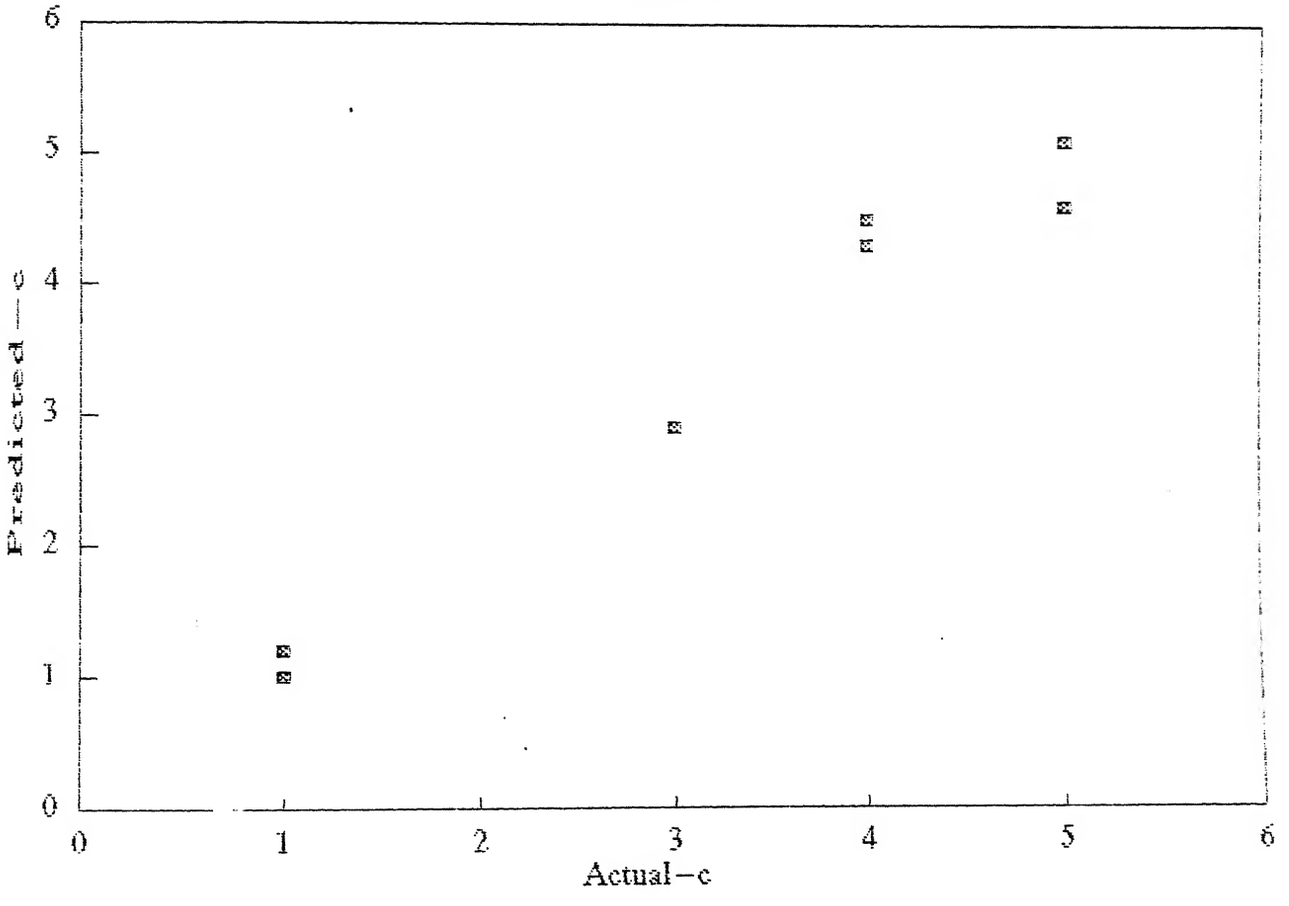


Table 7.3: 18–node ANN trained/tested on 75/10 patterns. (0.6/0.5).

Actual–n	Predctd–n	Error in n	% error in n	Actual–c	Predctd–c	Error in c	% error in c
41	37.36	−3.64	8.89	4	2.4	−1.6	39.92
47	50.29	3.29	7	3	4.76	1.76	58.78
27	26.26	−0.74	2.76	5	4.39	−0.61	12.21
43	43.49	0.49	1.15	2	1.87	−0.13	6.46
12	10	−2	16.69	5	3.62	−1.38	27.59
41	40.67	−0.33	0.8	0	−0.17	−0.17	
10	7.68	−2.32	23.2	1	−0.13	−1.13	112.55
43	41.47	−1.53	3.56	5	3.79	−1.21	24.16
11	6.72	−4.28	38.9	3	1.13	−1.87	62.32
40	32	−8	20.01	4	2.98	−1.02	25.56

Regression on n:

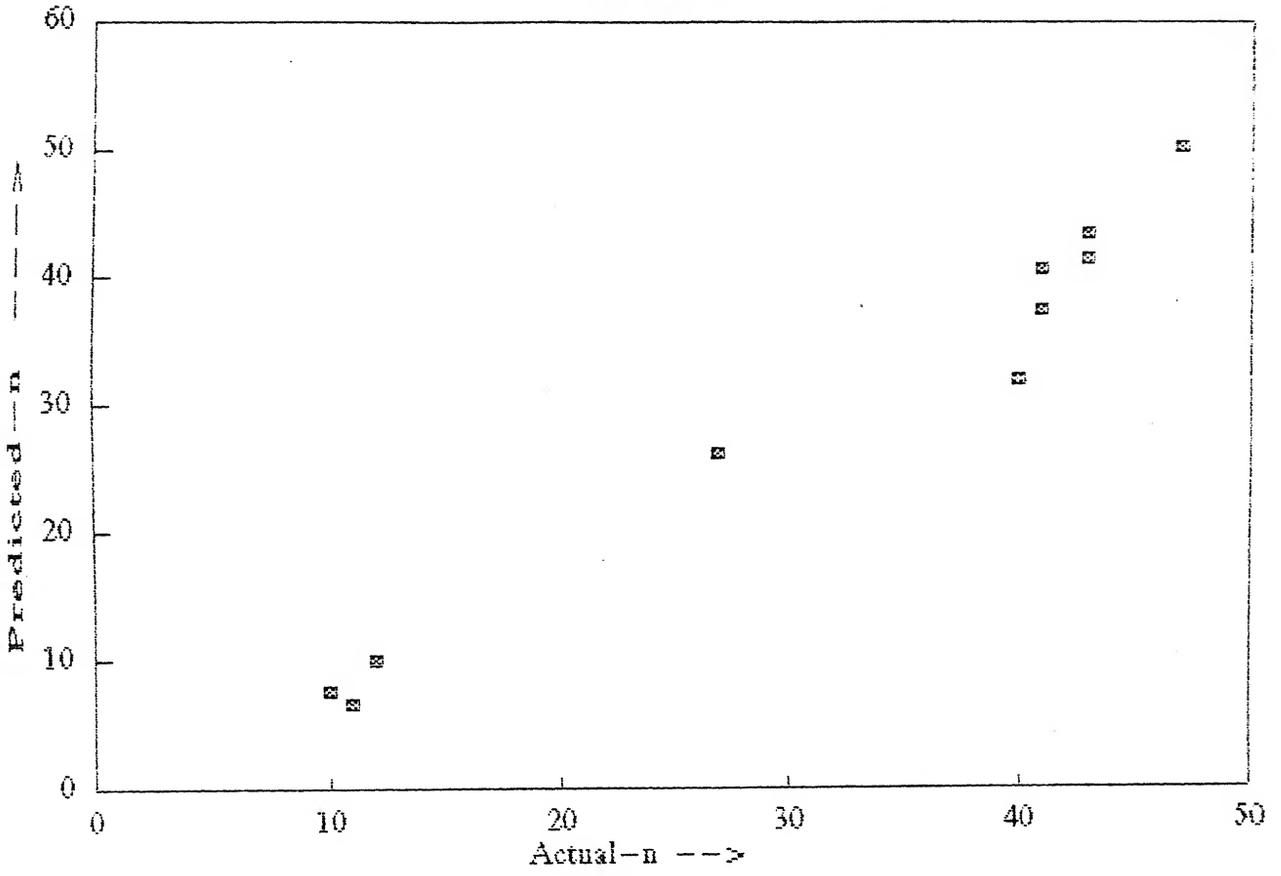
Constant	−3.65459
Std Err of Y Est	3.090225
R Squared	0.967487
No. of Observations	10
Degrees of Freedom	8
X Coefficient(s)	1.055511
Std Err of Coef.	0.06841

Regression on c:

Constant	−0.1971
Std Err of Y Est	1.064101
R Squared	0.678152
No. of Observations	10
Degrees of Freedom	8
X Coefficient(s)	0.831594
Std Err of Coef.	0.202548

SCATTER PLOT FOR n

in Table 7.3



SCATTER PLOT FOR c

in Table 7.3

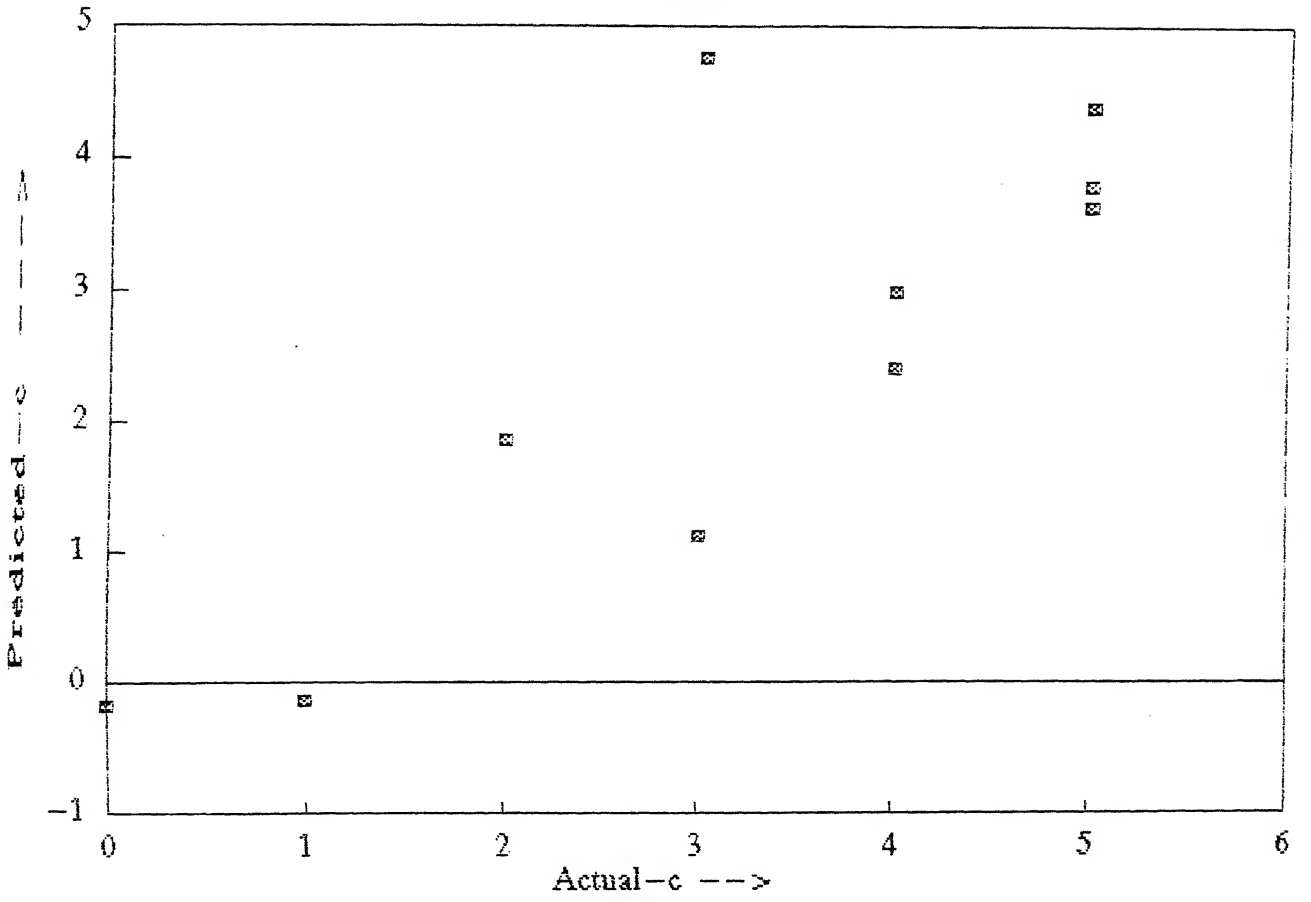
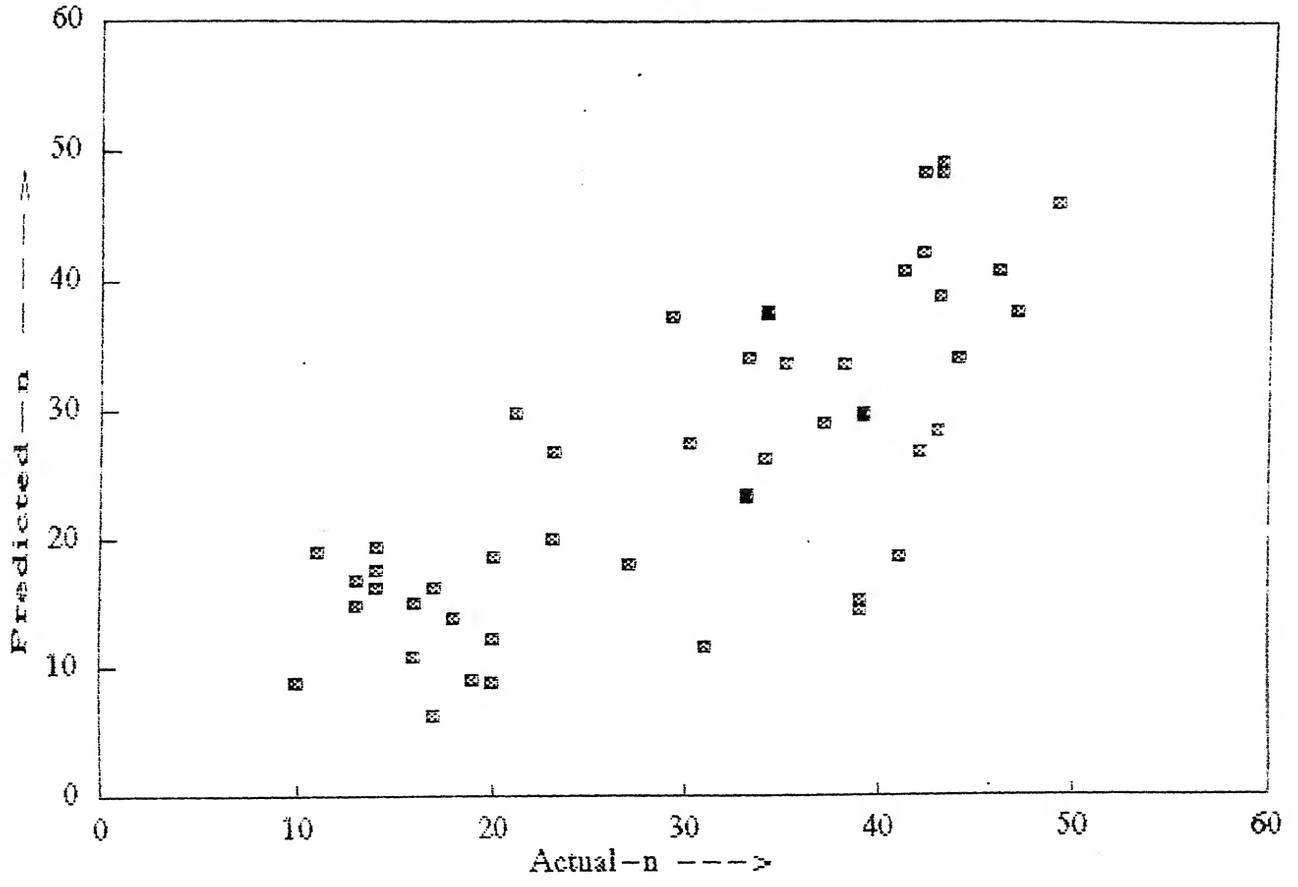


Table 7.4: 12–node ANN trained/tested on 50/50 patterns.

Actual–n	Predctd–n	Error in n	% error in n	Actual–c	Predctd–c	Error in c	% error in c
34	37.57	3.57	10.51	4	5.33	1.33	33.13
29	37.33	8.33	28.74	3	5.37	2.37	79.15
10	8.84	−1.16	11.6	4	5.05	1.05	26.15
42	42.3	0.3	0.71	4	4.74	0.74	18.39
13	14.9	1.9	14.59	3	3.86	0.86	28.82
39	29.71	−9.29	23.82	1	0.12	−0.88	88.13
43	49.12	6.12	14.23	0	−0.06	−0.06	
17	6.23	−10.77	63.36	0	−0.62	−0.62	
20	12.24	−7.76	38.82	4	2.9	−1.1	27.6
23	20.11	−2.89	12.58	4	3.92	−0.08	2.01
38	33.71	−4.29	11.28	3	2.96	−0.04	1.32
43	38.81	−4.19	9.75	4	3.77	−0.23	5.83
16	15.14	−0.86	5.38	3	3.58	0.58	19.18
20	18.6	−1.4	7	5	4.17	−0.83	16.58
33	23.17	−9.83	29.79	1	0.33	−0.67	67.31
42	48.47	6.47	15.41	3	3.78	0.78	25.9
49	45.94	−3.06	6.24	1	0.23	−0.77	77.24
37	29.14	−7.86	21.26	4	3.79	−0.21	5.14
27	17.98	−9.02	33.42	4	3.24	−0.76	18.92
16	10.93	−5.07	31.66	0	−0.01	−0.01	
20	8.92	−11.08	55.4	1	−0.47	−1.47	146.67
44	34.11	−9.89	22.47	0	−0.58	−0.58	
34	37.52	3.52	10.35	3	4.73	1.73	57.75
43	48.5	5.5	12.78	3	3.66	0.66	21.98
13	16.93	3.93	30.2	4	3.62	−0.38	9.46
17	16.26	−0.74	4.33	3	2.95	−0.05	1.61
39	14.53	−24.47	62.75	1	−0.49	−1.49	148.97
21	29.91	8.91	42.42	0	0.5	0.5	
46	40.9	−5.1	11.08	0	−0.36	−0.36	
41	40.74	−0.26	0.64	2	2.99	0.99	49.62
34	26.24	−7.76	22.83	4	3.95	−0.05	1.21
14	16.33	2.33	16.65	1	−0.02	−1.02	102.13
31	11.77	−19.23	62.04	3	−0.1	−3.1	103.47
33	23.51	−9.49	28.76	4	3.7	−0.3	7.6
18	13.91	−4.09	22.71	0	−0.1	−0.1	
47	37.66	−9.34	19.88	5	2.38	−2.62	52.49
39	15.22	−23.78	60.96	1	0.32	−0.68	67.76
39	29.76	−9.24	23.7	5	1.9	−3.1	62.07
33	34.06	1.06	3.2	2	3.22	1.22	60.81
14	19.38	5.38	38.4	4	4.6	0.6	15.02
23	26.9	3.9	16.97	1	2.29	1.29	129.37
43	28.45	−14.55	33.85	4	2.05	−1.95	48.84
14	17.71	3.71	26.53	5	3.91	−1.09	21.77
23	20.12	−2.88	12.53	0	0.13	0.13	
19	9	−10	52.64	1	0	−1	100.42
35	33.71	−1.29	3.68	5	4.39	−0.61	12.15
41	18.66	−22.34	54.49	0	−0.51	−0.51	
11	19.03	8.03	72.97	4	4.02	0.02	0.61
30	27.44	−2.56	8.53	2	3	1	50.21
42	26.85	−15.15	36.06	3	0.65	−2.35	78.39

SCATTER PLOT ON n

in Table 7.4



Regression on n:		Regression on c:	
Constant	2.826863	Constant	−0.10229
Std Err of Y Est	7.864872	Std Err of Y Est	1.167605
R Squared	0.567118	R Squared	0.655908
No. of Observations	50	No. of Observations	50
Degrees of Freedom	48	Degrees of Freedom	48
X Coefficient(s)	0.766037	X Coefficient(s)	0.935671
Std Err of Coef.	0.0966	Std Err of Coef.	0.097818

SCATTER PLOT FOR c

in Table 7.4